C++ für Java-Programmierer

Ein paar Stichpunkte und viele Beispielprogramme

von Ulrich Grude

Skript für Vorlesungen an der
Technischen Fachhochschule Berlin
Fachbereich Informatik (FB VI)
Version 0.5, WS05/06

Inhaltsverzeichnis

1 Literaturangaben.....5

2 Grundbegriffe der Programmierung.....7
2.1 Programmieren als ein Rollenspiel.....7
2.2 Die fünf wichtigsten Grundkonzepte von Programmiersprachen.....7
2.3 Drei Arten von Befehlen (des Programmierers an den Ausführer).....7
2.4 Größen benutzen.....7

3 Grundeigenschaften von C/C++.....8

4 C++ anhand von Beispielprogrammen.....9
4.1 Drei Hallo-Programme.....9
4.2 Zusammengesetzte Anweisungen.....10
4.3 Unterprogramme (Prozeduren, Funktionen, Operationen).....13
4.3.1 Vorbesetzungsausdrücke für die Parameter von Unterprogrammen.....13
4.3.2 Unterprogramme als Parameter von Unterprogrammen, ein Operator.....14
4.3.3 Unterprogramme mit einer variablen Anzahl von Parametern.....16
4.4 Ausnahmen (exceptions).....17
4.4.1 throw-, try-, catch-Anweisungen (aber kein finally!).....17
4.4.2 Ein Unterprogramm mit throw-Klausel.....18
4.4.3 Unerwartete Ausnahmen selbst abfangen.....19
4.5 Struktur und Ausführung eines C++-Programms.....21
4.6 Mehrere Quelldateien, ein Programm.....23
4.7 Compilieren und Binden, Deklarationen und Definitionen in Dateien.....24
4.8 Regeln über Deklarationen und Definitionen (in Dateien).....26
4.9 Daten mit printf formatieren und ausgeben.....31
4.10 Was bedeutet das Schlüsselwort static?.....34

5 Übersicht über alle Arten von C++-Typen (und ein Vergleich mit Java).....35

6 Die Bojendarstellung für Variablen.....40
6.1 Bojen für Variablen definierter Typen.....40
6.2 Bojen für zwei Adressvariablen.....41
6.3 Bojen für zwei Variablen mit gleichen Adressen und identischem Wert.....41
6.4 Bojen für zwei Variablen mit identischem Wert bzw. Zielwert.....42
6.5 Bojen für Konstanten, Adressen von Konstanten und konstante Adressen.....42
6.6 Boje für eine Reihung.....42
6.7 Bojen für Objekte.....43
6.8 Eine Adressvariable, die auf NanA (alias NULL) zeigt.....44

7 Konstruierte Typen.....45
7.1 Reihungstypen (array types).....49
7.1.1 Eindimensionale Reihungen.....50
7.1.2 Mehrdimensionale Reihungen.....52
7.2 C-Strings (Reihungen von char, Adressen von char).....56
7.3 Adresstypen und Referenztypen.....59

8 Variablen, Adresstypen und Referenztypen in C++.....59
8.1 Variablen, unveränderbare Variablen und Konstanten.....59
8.2 Adresstypen, Adressvariablen, Adresswerte und Adressen.....62
8.3 R-Werte und L-Werte.....63
8.4 Der Variablen-Operator * und der Adress-Operator &.....66
8.5 Mit Adresswerten rechnen.....69
8.6 Referenztypen.....73

9 Die Klassenschablone vector und die Klasse string.....77
9.1 vector-Objekte ("C++-Reihungen").....77
9.2 String-Objekte ("C++-Strings").....83

10 Generische Einheiten, Teil 1 (Unterprogrammschablonen, function templates).....87
10.1 Einleitung und Motivation.....87
10.2 Unterprogrammschablonen (function templates).....87

10.3 Der Unterschied zwischen Parametern und Parametern....88
10.4 Neue Möglichkeiten, Programmierfehler zu machen....89
10.5 Das Beispielprogramm Schablonen01 (die Schablone hoch)....89
10.6 Das Beispielprogramm Schablonen02 (mehrere Schablonen put)....91
10.7 Das Beispielprogramm Schablonen03 (die Schablone druckeVector)....93
10.8 Weitere Beispielprogramme mit Unterprogrammschablonen....94
10.9 Typparameter von Schablonen per Metaprogrammierung beschränken....94

11 Ein- und Ausgabe mit Strömen....97
11.1 Formatierte und unformatierte Ein-/Ausgabe....99
11.2 Verschiedene Befehle zum Einlesen von Daten....102
11.3 Eingabefehler beim formatierten Einlesen abfangen und behandeln....106
11.4 Zahlen beim Ausgeben formatieren (mit Strom-Methoden und Manipulatoren)....107
11.5 Zahlen formatieren ohne sie auszugeben....109

12 Klassen als Module und Baupläne für Module....111
12.1 Deklarationen und Definitionen von und in Klassen....114
12.2 Sichtbarkeitsregeln innerhalb einer Klasse....117
12.3 Eine Klasse Text "mit allem drum und dran"....118
12.4 Die Klasse Birnen, deren Objekte sich wie Zahlen verhalten....123
12.5 Einfaches Erben, virtuelle und nicht-virtuelle Methoden....126
12.6 Einfaches Erben, eine kleine Klassenhierarchie....129
12.7 Mehrfaches Erben....132
12.7.1 Mehrfaches Erben, ein einfaches (?) Beispiel....132
12.7.2 Mehrfaches Erben, virtuelle, überladene und überschriebene Methoden....134
12.7.3 Mehrfaches Erben, die Nappo-Frage....137
12.8 Klassen in C++ und in Java, wichtige Unterschiede....140

13 Generische Einheiten, Teil 2 (Klassenschablonen, class templates)....142
13.1 Die Klassenschablone Obst....142
13.2 Die Standard Template Library (STL)....145
13.3 Generische Einheiten in C++ und in Java, wichtige Unterschiede....146

14 Wiederverwendbare Teile im Vergleich: Module, Klassen, Schablonen....147
14.1 Der Stapel-Modul StapelM....147
14.2 Die Stapel-Klasse StapelK....151
14.3 Die Stapel-Schablone StapelS....153

15 Namensräume (namespaces)....156
15.1 Eigene Namensräume vereinbaren....156
15.2 Eigene Namensräume, weitere using-Direktiven und -Deklarationen....157
15.3 Deklarationen in einem Namensraum, Definitionen außerhalb....158
15.4 Geschachtelte Namensräume....158

16 Eine Schnittstelle zu einer Konsole unter Windows....160

17 Ein endlicher Automat....163
17.1 Problem....163
17.2 Endliche Automaten....163
17.3 Konkrete Eingabezeichen und abstrakte Eingaben....164
17.4 Der endliche Automat, dargestellt durch ein Diagramm (noch ohne Aktionen)....164
17.5 Die Aktionen des endlichen Automaten....165
17.6 Der endliche Automat, dargestellt durch eine Tabelle....165
17.7 Der endliche Automat in C++ programmiert....166

18 Höchste Zeit für eine Zeitmessung....168

19 Kommandozeilen-Befehle zum Compilieren und Binden....171
19.1 Gnu-Cygnus-Compiler g++....171
19.2 Borland C++Compiler bcc32....171
19.3 Microsoft C++-Compiler cl....172
19.4 Ein paar Skriptdateien, zum Compilieren und Binden einzelner Quelldateien....172
19.5 Eine make-Datei mit ein paar Erläuterungen....174

20 Sachwortverzeichnis..175

1 Literaturangaben

Zum Thema *C/C++* gibt es ***unüberschaubar*** viele Bücher. Hier ein kleine Auswahl:

Bjarne Stroustrup
The C++ Programming Language, Third Edition, Addison Wesley, 1997, ca. 50,- €.
Der Däne B. Stroustrup hat (in den USA) die Sprache C++ entwickelt. Seine Bücher gehören zum Besten, was es über C++ gibt. Er nimmt ***seine Sprache*** dem Leser gegenüber nicht in Schutz, sondern erläutert Höhen und Tiefen gleichermaßen genau und verständlich. Sehr empfehlenswert. Spätestens ab dem 3. Semester sollte jede StudentIn den Namen "Stroustrup" flüssig aussprechen können :-) !

David Vandevoorde
C++ Solutions, Addison Wesley 1998, 292 Seiten, ca. 26,- €.
Enthält Musterlösungen zu vielen Aufgaben aus dem Buch **The C++-Programming Language** von B. Stroustrup, sowie einige zusätzlichen Erläuterungen, Aufgaben und Lösungen. Empfehlenswert.

Bjarne Stroustrup
The Design and Evolution of C++, Addison Wesley, 1994, 461 Seiten, ca. 40,- €.
Warum ist die Sprache C++ so wie sie ist? Ein vor allem historisch sehr interessantes Buch. Nicht erforderlich, um die ersten C++-Programme zu schreiben, aber sehr interessant und wichtig für ein tieferes Verständnis der Sprache.

Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language, Second Edition, Prentice Hall 1988, 272 Seiten, ca. 45,- €.
Die Sprache C++ ist eine Erweiterung von C. Dies ist eines der besten Bücher über C, von den Entwicklern der Sprache selbst. Es gibt wohl auch eine deutsche Übersetzung.

Ulrich Breymann
C++, Eine Einführung, 8. erweiterte Auflage, Hanser-Verlag 2005, ca. 40,- €.
Ein ordentliches Buch über C++ auf Deutsch (eine 8. Auflage ist ein sehr, sehr, sehr gutes Zeichen).

Mark Allen Weiss
C++ for Java Programmers, Pearson Education, 280 Seiten, ca. 42,- €
Ein relativ kurzes Buch, in dem alle wichtigen Unterschiede zwischen Java und C++ recht gut erklärt werden. An einigen Stellen muss man selbst klar zwischen Typen und Variablen bzw. zwischen Klassen und Objekten unterscheiden, weil der Autor das nicht tut.

Nicolai M. Josuttis
The C++ Standard Library, Addison Wesley 1999, 800 Seiten, ca. 55,- €.
Seit 1998 gibt es einen ISO-Standard für C++. Dieser Standard schreibt auch eine ***Bibliothek*** vor, zu der vor allem ein System von Behälterschablonen (entsprechen in etwa den generischen Java-Samnlungsklassen) und Algorithmen zum Bearbeiten von Behältern gehören. Ein ernsthafter C++-Programmierer braucht (mindestens) ein Buch über diese Standardbibliothek. Das Buch von Josuttis (aus Braunschweig) ist auf englisch, aber besonders verständlich geschrieben und auch als Nachschlagewerk gut geeignet. Relativ teuer, trotzdem sehr empfehlenswert.

Stefan Kuhlins, Martin Schader
Die C++-Standardbibliothek, Springer-Verlag 1999, 384 Seiten, ca. 35,- .

Dieses Buch ist auf Deutsch und relativ billig.

S. B. Lippman, J. Lajoie
C++ Primer, 4th Edition, Addison-Wesley 2005, ca. 885 Seiten, ca. 38,- €.
Ein sehr gutes amerikanisches Lehrbuch über C++ (eine 4. Auflage ist ein ***sehr gutes*** Zeichen), mit vielen Beispielen, Aufgaben und sehr gut verständlichen Erklärungen. Vor allem die ***schwierigen Stellen*** in C++ werden sehr genau und ***verständlich*** beschrieben. Der Preis entspricht dem großen Umfang. Ein Buch mit Lösungen zu den Aufgaben ist ebenfalls im Buchhandel erhältlich (siehe nächstes Buch).

C. L. Tondo, B. P. Leung
C++ Primer Answer Book, Addison-Wesley 1999, 430 Seiten, ca. 39,- €
Lösungen zu den Aufgaben im Buch **C++ Primer** von Lippman und Lajoie, mit guten Erläuterungen dazu.

H.M. Deitel, P.J. Deitel (Vater und Sohn)
Small C++, How to Program, 5th Edition, Prentice Hall 2005, 700 Seiten, ca. 74,- €
Ebenfalls ein sehr gutes amerikanisches Lehrbuch über C++ (eine 5. Auflage ist ein ***gutes*** Zeichen), mit vielen Beispielen, Aufgaben und sehr gut verständlichen Erklärungen. Der Preis entspricht dem großen Umfang. Ein ***Instructors's Manual*** (mit Lösungen zu allen Aufgaben) ist nur direkt vom Verlag und gegen den Nachweis einer Lehrtätigkeit erhältlich.

Die Firma Borland (bzw. Inprise) stellt im Internet allen Lernenden (die ein Windows Betriebssystem benutzen) nicht nur einen sehr guten C++-Compiler umsonst zur Verfügung (BCC Version 5.5), sondern auch recht gute Hilfe-Dateien dazu. Insbesondere die Datei `BCB5LANG.HLP` beschreibt wichtige Eigenschaften der Sprache C++ weitgehend unabhängig von den Eigenheiten des Borland-Compilers. Siehe URL `www.borland.com/downloads/download_cbuilder.html`.

R. Lischner
C++ in a Nutshell, O'Reilly 2003, 790 Seiten, ca. 44 €
Etwa 270 Seiten Darstellung der Sprache und 520 Seiten Nachschlagewerk. Empfehlenswert.

The C++ Standard, British Standard (BS), International Organisation for Standardization (ISO) / International Electrotechnical Commission (IEC) 14882:2003 (Second Edition)
Verlag John Wiley & Sons, 782 Seiten, ca. 55,- €
Schwer lesbar, nicht besonders gut strukturiert. Compiler-Bauer ***müssen*** sich mit diesem Dokument auseinandersetzen, Programmierer ***vermeiden*** es in aller Regel eher. Mal einen Blick hineinzuwerfen (z. B. in der TFH-Bibliothek oder in der Buchhandlung Lehmanns) kann aber interessant sein.

Von der folgenden Adresse kann man **The C++Standard** (ISO/IEC 14882) herunterladen (PDF-Datei, ca. 2,7 MB, kostet 18,- US$): `www.techstreet.com/ncits.html`

Die quelloffene und interessante Entwickulungsumgebung **Dev-C++** (für die Entwicklung von C++-Programmen) kann man von folgender Adresse herunterladen: `www.bloodshed.net`.

Die **Gnu-Cygnus-Werkzeuge** (Linux-Werkzeuge unter Windows, einschliesslich der Gnu Compiler Collection `gcc/g++`) findet man unter `www.cygwin.com`.

2 Grundbegriffe der Programmierung

2.1 Programmieren als ein Rollenspiel

Die 5 Rollen und ihre charakteristischen Tätigkeiten:
Programmierer: Schreibt Programme, übergibt sie dem Ausführer
Ausführer: Prüft Programme, lehnt sie ab oder akzeptiert sie, führt Programme aus (auf Befehl des Benutzers)
Benutzer: Befiehlt dem Ausführer, Programme auszuführen. Ist für Ein- und Ausgabedaten zuständig.
Kollege 1: Verbessert, verändert, erweitert (kurz: wartet) die Programme des Programmierers.
Kollege 2: Ist daran interessiert, Programme des Programmierers wiederzuverwenden.

2.2 Die fünf wichtigsten Grundkonzepte von Programmiersprachen

Variablen (Wertebehälter, deren Wert man beliebig oft verändern kann)
Typen (Baupläne für Variablen)
Unterprogramme (andere Bezeichnungen: Funktionen, Prozeduren, Methoden, ...)
Module (Behälter für Variablen, Unterprogramme, weitere Module etc., mit einem sichtbaren, ungeschützten und einem unsichtbaren, geschützten Teil)
Klassen (Eine Klasse ist ein Modul und ein Bauplan für Module, die nach dem Bauplan gebauten Module nennt man auch ***Objekte***)

2.3 Drei Arten von Befehlen (des Programmierers an den Ausführer)

Vereinbarung (declaration, "Erzeuge ...", z. B. eine Variable, ein Unterprogramm, einen Typ, ...)
Ausdruck (expression, "Berechne den Wert des Ausdrucks ...")
Anweisung (statement, "Tue die Werte ... in die Wertebehälter ...").
Eine besonders wichtige Anweisung ist die Zuweisungsanweisung.

2.4 Größen benutzen

Als ***Größen*** werden in diesem Skript alle Dinge bezeichnet, die der Programmierer in einem Programm ***vereinbaren*** (d. h. vom Ausführer erzeugen lassen) kann. Die wichtigsten Größen sind ***Variablen***, ***Konstanten***, ***Unterprogramme***, ***Typen*** (insbesondere Klassentypen). ***Werte*** zählen ***nicht*** zu den Größen, da man sie nicht vereinbaren kann (Werte werden ***berechnet***, nicht erzeugt). Nachdem man eine ***Größe*** vereinbart hat, kann man sie ***benutzen***. ***Variablen*** und ***Konstanten*** kann man ***als Ausdrücke*** oder ***als Teile eines Ausdrucks*** benutzen, Variablen kann man außerdem als Ziel einer Zuweisung (oder einer anderen Anweisung) benutzen, ***Unterprogramme*** kann man ***aufrufen***, ***Typen*** kann man dazu benutzen, ***Variablen*** und ***Konstanten*** zu vereinbaren.

3 Grundeigenschaften von C/C++

1. Die Entwicklung der Sprache ***C*** begann etwa ***1970***, die Entwicklung von ***C++*** etwa ***1980***. ***C++*** ist eine ***Erweiterung*** von ***C***. Mit ganz wenigen Ausnahmen ist jedes ***C***-Programm auch ein ***C++***-Programm.

2. Zahlreiche Assembler-Programmierer empfanden (ab etwa 1960) die ersten Fortran-Compiler der "weltbeherrschenden" Firma IBM nicht als ***Hilfe***, sondern als ***Einschränkung ihrer Freiheit beim Programmieren*** ("Big-Brother-Syndrom"). ***C*** sollte Programmierer unterstützen, aber ihre Freiheit möglichst ***nicht einschränken***.

3. Ein wichtiger Grundsatz bei der Entwicklung von ***C***: Der Programmierer ***weiss was er tut***. Ein ***C-Compiler*** soll möglichst nichts ***verbieten***, was der ***Programmierer*** für ***sinnvoll hält***.

4. Ein wichtiger Grundsatz bei der Entwicklung von ***C++***: Die ***Effizienz*** der ***Programme*** ist wichtiger als die ***Sicherheit*** der ***Sprache***. Das gilt insbesonders auch für die neusten Teile von C++ (die Standard Template Library STL). Begründung: In ein ***schnelles*** Programm kann man zusätzliche Prüfungen einbauen, um seine ***Sicherheit*** zu ***erhöhen***. Wenn aber ein Programm ***langsam*** ist, weil die verwendete Programmiersprache viele Prüfungen vorschreibt und einbaut, kann man es kaum schneller machen.

5. Seit 1998 gibt es einen internationalen (ISO/IEC-) Standard für C++ (siehe oben die Literaturangaben dazu), was sehr zu begrüßen ist. Man sollte die Wirkung dieses Standards aber nicht überschätzen. So legt er z. B. nur eine Reihe von Typnamen fest (`char`, `short`, `int`, `unsigned int`, `long`, `float` etc.), läßt aber ausdrücklich offen, wie viele Typen diese Namen bezeichnen (ob z. B. die Namen `short` und `int` zwei verschiedene Typen oder denselben Typ bezeichnen), welche Werte zu den einzelnen Typen gehören und welche Ergebnisse die Funktionen und Operationen (z. B. `+`, `/`, `%` etc.) der Typen liefern. Der Standard schreibt auch eine umfangreiche Standardbibliothek vor, die vor allem Behälterschablonen enthält und Algorithmen, die Behälter bearbeiten (C++-Behälterschablonen entsprechen in etwa den generischen Sammlungsklassen in Java und C++-Behälter den Java-Sammlungen). Der C++-Standard schreibt aber nur relativ schlichte Ein-/Ausgabe-Befehle vor. Wenn man z. B. Grabos (graphische Benutzeroberflächen, engl. GUIs) oder Netzanwendungen programmieren will, muss man zusätzliche, nicht-standardisierte Bibliotheken benutzen.

6. Kritische Fragen: Ist die Effizienz einer Programmiersprache (noch) so wichtig, wie die Entwickler von C und C++ annahmen? Heute übliche Prozessoren sind etwa eine Million mal schneller als 1970 verbreitete Prozessoren. Was ist mit der ***Effizienz*** der ***Programmierer***? Was ist mit der ***Effizienz*** der ***Programmbenutzer***? Wie viele der zahlreichen ***Fehler*** in verbreiteten Programmen hätte man durch Verwendung einer sichereren Sprache ***vermeiden*** können? Was hätte die Verwendung einer sichereren Sprache ***gekostet*** (an Geschwindigkeit, Speicher, Geld)? Wissen Programmierer (immer? meistens? manchmal?) was sie tun?

4 C++ anhand von Beispielprogrammen

4.1 Drei Hallo-Programme

Ausgabe zur Standardausgabe `cout` (character output), `#include` ohne `using`, verschiedene Formen von Kommentaren:

```
1   // Datei Hallo01.cpp
2
3   #include <iostream> // fuer cout und endl
4
5   int main() {
6      std::cout << "Hallo, hier ist das Programm Hallo01!" << std::endl;
7   } // main
8   /* ------------------------------------------------------------------------
9   Ausgabe des Programms Hallo01:
10
11  Hallo, hier ist das Programm Hallo01!
12  ------------------------------------------------------------------------ */
```

In Zeile 6 bezeichnet `std` einen Namensraum (namespace). Namensräume in C++ haben Ähnlichkeit mit Paketen in Java.

Eine `main`-Funktion liefert einen `int`-Wert an das Betriebssystem, `#include` mit `using`-*Deklarationen*:

```
1   // Datei Hallo02.cpp
2
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   int main() { // Eine main-Funktion (keine main-Prozedur!)
8      cout << "Hallo, hier ist das Programm Hallo02!" << endl;
9      return 0; // Nachricht an das Betriebssystem: "Das Programm Hallo02
10               // hat sich beendet ohne dass ein Fehler auftrat".
11  } // main
```

`#include` mit `using`-*Direktive*, Variablen *vereinbaren* ("nach den Bauplänen `int` bzw. `string` bauen lassen") und *initialisieren*, String-Literale über Zeilengrenzen hinweg notieren, Strings konkatenieren, `string`- und `int`-Werte einlesen:

```
1   // Datei Hallo03.cpp
2
3   #include <iostream> // fuer cout und endl
4   #include <string>   // fuer string
5   using namespace std;
6
7   int main() {
8      string meldung1 = "Bitte geben Sie Ihren Namen " // ohne newline
9                        "und Ihr Alter ein: ";
10     string meldung2 = "Gut, dass dieses Programm\n"  // mit  newline
11                       "jugendfrei ist!";
12     string name;
13     int    alter;
14
15     cout << meldung1;
16     cin  >> name >> alter; // Einen String und eine Ganzzahl einlesen
17
18     cout << "Guten Tag, " << name << "!"    << endl;
19     cout << alter << " Jahre? " + meldung2  << endl;
```

```
20
21  } // main
22  /* ------------------------------------------------------------------------
23  Ein Dialog mit dem Programm Hallo03 (Eingaben des Benutzers fett):
24
25  Bitte geben Sie Ihren Namen und Ihr Alter ein: Heinrich 22
26  Guten Tag, Heinrich!
27  22 Jahre? Gut, dass dieses Programm
28  jugendfrei ist!
29  ------------------------------------------------------------------------ */
```

Ähnlich wie hier mit `string`- und `int`-Variablen umgegangen wurde, kann man auch mit Variablen der Typen `char`, `short`, `long`, `unsigned int`, `unsigned long`, `float`, `double`, `long double`, `bool` etc. umgehen.

Hinweis: Einige Typnamen wie `unsigned int` und `long double` etc. bestehen jeweils aus ***zwei*** Worten! Statt `unsigned int` kann man auch einfach `unsigned` schreiben.

Das Programm `Hallo04` (hier nicht wiedergegeben) zeigt, wie man auf ***Kommandozeilen-Parameter*** zugreift.

4.2 Zusammengesetzte Anweisungen

Eine ***einfache Anweisung*** kann Namen und Ausdrücke enthalten, aber keine anderen ***Anweisungen***. Beispiele für einfache Anweisungen: Die Zuweisungsanweisung (z. B. `x = y + 1;`), die return-Anweisung (z. B. `return;` oder `return x + 17;`) und jeder Aufruf einer Prozedur.

Eine ***zusammengesetzte Anweisung*** enthält unter anderem andere ***Anweisungen***. Beispiele für zusammengesetzte Anweisungen: `if`-, `switch`-, `while`-, `do-while`-, `for`-Anweisung.

Enthält ein Unterprogramm nur ***einfache Anweisungen***, dann wird bei jeder Ausführung des Unterprogramms jede Anweisung ***genau einmal*** ausgeführt. Mit ***zusammengesetzten Anweisungen*** kann der Programmierer bewirken, dass bestimmte Anweisungen seines Programms unter bestimmten Bedingungen ***weniger als einmal*** (d. h. nicht) bzw. ***mehr als einmal*** ausgeführt werden. Genauer: Mit `if`- und `switch`-Anweisungen lässt man die darin enthaltenen Anweisungen (unter bestimmten Bedingungen) ***weniger als einmal*** ausführen und mit Schleifen (`while`-, `do-while`- und `for`-Anweisungen) bewirkt man typischerweise, dass die darin enthaltenen Anweisungen ***mehr als einmal*** ausgeführt werden.

```
1   // Datei IfSwitch01.cpp
2   // ------------------------------------------------------------------------
3   // if- und switch-Anweisungen, einfache Beispiele
4   // ------------------------------------------------------------------------
5   #include <iostream>
6   #include <climits>  // fuer UCHAR_MAX und CHAR_MAX
7   using namespace std;
8
9   int main() {
10     cout << "Programm IfSwitch01: Jetzt geht es los! "        << endl;
11     // ---------------------------------------------------------------------
12     if (UCHAR_MAX != CHAR_MAX) {
13        cout << "UCHAR_MAX und CHAR_MAX sind verschieden!"    << endl;
14        cout << "UCHAR_MAX: " << UCHAR_MAX <<
15                " CHAR_MAX: " <<  CHAR_MAX << endl;
16     }
17     // ---------------------------------------------------------------------
18     cout << "Bitte eine Ganzzahl eingeben: ";
19     int eingabe;
20     cin  >> eingabe;
```

```
21     // ------------------------------------------------------------------------
22     if (eingabe % 2) {
23        cout << "Sie haben eine ungerade Zahl eingegeben!"    << endl;
24     } else {
25        cout << "Sie haben eine gerade Zahl eingegeben!"      << endl;
26     } // if
27     // ------------------------------------------------------------------------
28     if (eingabe == 0) {
29        cout << "Sie haben die Zahl 0 eingegeben!"            << endl;
30     } else if (1 <= eingabe && eingabe <= 100) {
31        cout << "Ihre Zahl liegt zwischen 1 und 100!"         << endl;
32     } else if (eingabe < 200 || 300 < eingabe) {
33        cout << "Ihre Zahl liegt nicht zwischen 200 und 300!" << endl;
34     } else {
35        cout << "Ihre Zahl liegt zwischen 200 und 300!"       << endl;
36     } // if
37     // ------------------------------------------------------------------------
38     switch (eingabe) {
39        case 0:
40           cout << "Sie haben die Zahl 0 eingegeben!"         << endl;
41           break;
42        default:  // "default" muss nicht zuletzt stehen!
43           cout << "Sie haben eine komische Zahl eingegeben!" << endl;
44           break;
45        case 2:
46        case 3:
47        case 4:
48        case 5:
49           cout << "Ihre Eingabe liegt zwischen 2 und 5!"     << endl;
50           break; // Nicht noetig, aber aenderungsfreundlich!
51     } // switch
52     // ------------------------------------------------------------------------
53     cout << "Programm IfSwitch01: Das war's erstmal!"        << endl;
54  } // main
55  /* ------------------------------------------------------------------------
56  Ein Dialog mit dem Programm IfSwitch01 (Eingabe des Benutzers fett):
57
58  Programm IfSwitch01: Jetzt geht es los!
59  UCHAR_MAX und CHAR_MAX sind verschieden!
60  UCHAR_MAX: 255 CHAR_MAX: 127
61  Bitte eine Ganzzahl eingeben: 123
62  Sie haben eine ungerade Zahl eingegeben!
63  Ihre Zahl liegt nicht zwischen 200 und 300!
64  Sie haben eine komische Zahl eingegeben!
65  Programm IfSwitch01: Das war's erstmal!
66  ------------------------------------------------------------------------ */
```

Hinweis: In einer `switch`-Anweisung darf (nach dem Schlüsselwort `switch` in runden Klammern) ein *beliebig (einfacher oder) komplizierter Ausdruck* angegeben werden, z. B. `eingabe` oder `2*eingabe+17` etc. Dieser Ausdruck muss zu einem "kleinen" *Ganzzahltyp* gehören (`char`, `short`, `int`, `unsigned char`, `unsigned short`, `unsigned int`, `bool` (!)), darf aber nicht zum Typ `long` oder `unsigned long` gehören. Die `default`-Alternative in einer `switch`-Anweisung muss *nicht* unbedingt an *letzter* Stelle stehen.

```
1   // Datei Schleifen01.cpp
2   // ------------------------------------------------------------------------
3   // for-, while- und do- while-Schleifen, einfache Beispiele
4   // ------------------------------------------------------------------------
5   #include <iostream>
6   using namespace std;
```

```
7
8   int main() {
9      cout << "Programm Schleifen01: Jetzt geht es los!" << endl;
10     // ------------------------------------------------------------------
11     short exponent;
12     cout << "Bitte einen (ganzzahligen) Exponenten eingeben: ";
13     cin  >>  exponent;
14     // ------------------------------------------------------------------
15     short ergebnis1 = 1;
16     for (int i=1; i<=exponent; i++) { // i innerhalb der Schleife vereinbart
17        ergebnis1 *= 2;
18        int j = 777;                 // j innerhalb der Schleife vereinbart
19     }
20     cout << "ergebnis1: " << ergebnis1 << endl;
21  // cout << "i        : " << i         << endl; // Warnung oder Fehlermeld.
22  // cout << "j        : " << j         << endl; // Warnung oder Fehlermeld.
23     // ------------------------------------------------------------------
24     int ergebnis2 = 1;
25     int i         = 1;                 // i ausserhalb der Schleife vereinb.
26     for ( ; i <=exponent; i++) {
27        ergebnis2 *= 2;
28     }
29     cout << "ergebnis2: " << ergebnis2 << endl;
30     cout << "i        : " << i         << endl; // Keine Warnung, alles ok.
31     // ------------------------------------------------------------------
32     long ergebnis3 = 1;
33     for (int i=exponent ; i; i--) {
34        ergebnis3 *= 2;
35     }
36     cout << "ergebnis3: " << ergebnis3 << endl;
37     // ------------------------------------------------------------------
38     long double ergebnis4 = 1.0;
39     int         k         = exponent;
40     do {
41        ergebnis4 *= 2;
42     } while (--k);
43     cout.setf(ios::fixed);
44     cout << "ergebnis4: " << ergebnis4 << endl;
45     // ------------------------------------------------------------------
46     cout << "Programm Schleifen01: Das war's erstmal!" << endl;
47  } // main
48  /* ------------------------------------------------------------------------
49  Fehlermeldung des Compilers, falls Zeile 21 und 22 keine Kommentare sind:
50
51  Schleifen01.cpp:21: name lookup of `i' changed for new ANSI `for' scoping
52  Schleifen01.cpp:16:   using obsolete binding at `i'
53  Schleifen01.cpp:22: `j' undeclared (first use this function)
54  ---------------------------------------------------------------------------
55  Ein Dialog mit dem Programm Schleifen01:
56
57  Programm Schleifen01: Jetzt geht es los!
58  Bitte einen (ganzzahligen) Exponenten eingeben: 10
59  ergebnis1: 1024
60  ergebnis2: 1024
61  i        : 11
62  ergebnis3: 1024
63  ergebnis4: 1024.000000
64  Programm Schleifen01: Das war's erstmal!
65  ---------------------------------------------------------------------------
66  Noch ein Dialog mit dem Programm Schleifen01:
67
68  Programm Schleifen01: Jetzt geht es los!
```

```
69  Bitte einen (ganzzahligen) Exponenten eingeben: 31
70  ergebnis1: 0
71  ergebnis2: -2147483648
72  i        : 32
73  ergebnis3: -2147483648
74  ergebnis4: 2147483648.000000
75  Programm Schleifen01: Das war's erstmal!
76  ------------------------------------------------------------------------ */
```

Hinweis 1: Einige der Ausgaben im zweiten Dialog mit dem Programm `Schleifen01` sind ***falsch*** (`2` hoch `31` ist ja nicht gleich `0` oder gleich `-2147483648`). Damit wird eine ***schreckliche*** Eigenschaft der Sprache C++ sichtbar: Beim Rechnen mit Ganzzahlen lösen ***Überläufe*** keine Ausnahme aus und führen auch nicht immer zu einem ***offensichtlich falschen*** Ergebnis.

Hinweis 2: Eine Variable, die ***innerhalb einer Schleife*** vereinbart wird, lebt nur während der Ausführung dieser Schleife.

Das Programm `Schleifen02` (hier nicht wiedergegeben) enthält Beispiel für Schleifen mit `break`- und `continue`-Anweisungen und Schleifen "im C-Stil".

4.3 Unterprogramme (Prozeduren, Funktionen, Operationen)

Unterprogramme in ***C++*** haben große ***Ähnlichkeit*** mit Methoden in ***Java***, aber es gibt auch ein paar Unterschiede.

Unterschied 1: In C++ kann man beim Vereinbaren eines Unterprogramms für alle oder einige Parameter ***Vorbesetzungsausdrücke*** (default arguments) festlegen. Beim Aufrufen des Unterprogramms muss man dann für diese Parameter nicht unbedingt einen aktuellen Wert angeben (man kann aber). Einen ähnlichen Effekt kann man in Java nur dadurch erreichen, dass man ***mehrere*** Unterprogramme (mit ***gleichen*** Namen und ***unterschiedlichen*** Parameterlisten) vereinbart (was meist sehr viel umständlicher ist als die C++-Lösung mit Vorbesetzungen).

Unterschied 2: In C++ kann man einem Unterprogramm ohne weiteres Unterprogramme als Parameter übergeben. In Java muss man Unterprogramme erst ***in ein Objekt einwickeln***, ehe man sie als Parameter übergeben kann.

Unterschied 3: In Java kann der Programmierer nur Unterprogramme mit ***normalen Namen*** wie `sinus` oder `meine_summe_17` vereinbaren. In C++ kann der Programmierer auch Unterprogramme mit Namen wie `+`, `-`, `*`, `/`, `[]`, `()` etc. vereinbaren. Solche Unterprogramme werden hier als ***Operationen*** und ihre Namen als ***Operatoren*** bezeichnet. Die meisten Operatoren sind ***überladen***, d. h. ein Operator bezeichnet mehrere verschiedene Operationen (z. B. bezeichnet der Operator + mehrere Ganzzahl- und Bruchzahloperationen).

4.3.1 Vorbesetzungsausdrücke für die Parameter von Unterprogrammen

Hier ein Beispielprogramm zum Thema Vorbesetzungsausdrücke (default arguments):

```
1   // Datei Upro01.cpp
2   /* ------------------------------------------------------------------------
3   Das Unterprogramm add hat 5 Parameter. Davon sind 3 mit Vorbesetzungs-
4   ausdruecken versehen. Somit kann man add wahlweise mit 2, 3, 4 oder 5
5   Parametern aufrufen.
6   ------------------------------------------------------------------------ */
7   #include <iostream>
8   using namespace std;
9   // ------------------------------------------------------------------------
```

```
10 int add(int i1, int i2, int i3=0, int i4=0, int i5=0) {
11    // Liefert die Summe der Zahlen i1 bis i5
12    return i1 + i2 + i3 + i4 + i5;
13 } // add
14 // ------------------------------------------------------------------------
15 int main() {
16    cout << "Upro01: Jetzt geht es los!"                   << endl;
17    cout << "add(3, 5):          " << add(3, 5)            << endl; // 2 Param.
18    cout << "add(3, 5, 2, 9, 4): " << add(3, 5, 2, 9, 4) << endl; // 5 Param.
19 // cout << "add(3)              " << add(3)               << endl; // 1 Param.
20    cout << "Upro01: Das war's erstmal!"                   << endl;
21 } // main
22 /* ------------------------------------------------------------------------
23 Fehlermeldung des Compilers, wenn Zeile 19 kein Kommentar ist:
24
25 Upro01.cpp:10: too few arguments to function
26    `int add(int, int, int = 0, int = 0, int = 0)'
27 Upro01.cpp:19: at this point in file
28 ------------------------------------------------------------------------
29 Ausgabe des Programms Upro01:
30
31 Upro01: Jetzt geht es los!
32 add(3, 5):          8
33 add(3, 5, 2, 9, 4): 23
34 Upro01: Das war's erstmal!
35 ------------------------------------------------------------------------ */
```

Hinweis: Unterprogramme wie `Upro01.cpp` (bei denen der Programmierer für einige Parameter ***Vorbesetzungsausdrücke*** angegeben hat) gibt es nur in C und C++, aber nicht in Java 5 (vielleicht werden sie mit der nächsten Java-Version eingeführt?). Unterprogramme mit einer ***variablen Anzahl von Parametern*** gibt es in C/C++ und in Java 5 (siehe dazu den folgenden Abschnitt).

Aufgabe: Schreiben Sie eine Funktion namens `max` mit `int`-Ergebnis, die man wahlweise mit 2, 3, 4 oder 5 `int`-Parametern aufrufen kann und die jeweils ***den größten ihrer Parameter*** als Ergebnis liefert. Hinweis: Die Konstante `INT_MIN` wird in der Kopfdatei `<climits>` definiert und bezeichnet den kleinsten `int`-Wert des Ausführers.

4.3.2 Unterprogramme als Parameter von Unterprogrammen, ein Operator

Das folgende Beispielprogramm enthält ein Unterprogramm, dem man ein ***Unterprogramm als Parameter*** übergeben kann. Außerdem enthält es eine selbst definierte Operation (d. h. eine Funktion mit einem Operator als Namen):

```
1  // Datei Upro02.cpp
2  /* ------------------------------------------------------------------------
3  Demonstriert ein Sortier-Unterprogramm (sortVI), dem man ein Unterprogramm
4  als Parameter uebergeben kann. Ausserdem wird ein Operator vereinbart.
5  ------------------------------------------------------------------------ */
6  #include <vector>
7  #include <iostream>
8  using namespace std;
9  // ------------------------------------------------------------------------
10 // Zum Funktions-Typ VergleichsFunktion gehoeren alle Funktionen mit
11 // einem bool-Ergebnis und zwei int-Parametern:
12 typedef bool VergleichsFunktion(int, int);
13
14 // Die folgenden beiden Funktionen gehoeren zum Typ Vergleichsfunktion:
15 bool  kleinerGleich(int i1, int i2) {return i1 <= i2;}
16 bool groesserGleich(int i1, int i2) {return i1 >= i2;}
```

```
17 // ------------------------------------------------------------------------
18 // Eine Sortier-Funktion, der man (ausser dem zu sortierenden int-Vector)
19 // eine VergleichsFunktion als Parameter uebergeben kann.
20 // Sortiert wird nach der Methode "Wiederholtes Vertauschen benachbarter
21 // Komponenten" (bubblesort).
22 vector<int>    // Ergebnistyp der Funktion sortVI
23 sortVI(vector<int> v, VergleichsFunktion liegenRichtig = kleinerGleich) {
24    int tmp; // Zum Vertauschen zweier int-Komponenten
25    if (v.size() <= 1) return v;
26    for (vector<int>::size_type i=v.size()-1; i>=1; i--) {
27       for (vector<int>::size_type j=0; j<i; j++) {
28          if (!liegenRichtig(v[j], v[j+1])) {
29             // v[j] und v[j+1] vertauschen
30             tmp    = v[j];
31             v[j]   = v[j+1];
32             v[j+1] = tmp;
33          }
34       } // for j
35    } // for i
36    return v;
37 } // sortVI
38 // ------------------------------------------------------------------------
39 // Ein Ausgabeoperator fuer int-Vectoren:
40 ostream & operator << (ostream & os, vector<int> v) {
41    cout << "Ein int-Vector der Laenge " << v.size() << ": ";
42    for (vector<int>::size_type i=0; i<v.size(); i++) {
43       cout << v[i] << " ";
44    }
45    return os;
46 } // operator <<
47 // ------------------------------------------------------------------------
48 int main() {
49    // Kleiner Test der Sortierfunktion sortVI und des Ausgabeoperators
50    // "<<" fuer int-Vectoren:
51
52    cout << "Upro02: Jetzt geht es los!"       << endl;
53
54    // Einen int-Vector vereinbaren und initialisieren (nicht sehr schoen):
55    vector<int> v1(5);
56    v1[0]=49; v1[1]=32; v1[2]=63; v1[3]=17; v1[4]=54;
57
58    // Den int-Vector mehrmals sortieren und ausgeben:
59    cout << sortVI(v1,  kleinerGleich)    << endl;
60    cout << sortVI(v1)                    << endl;
61    cout << sortVI(v1, groesserGleich)    << endl;
62
63    cout << "Upro02: Das war's erstmal!"  << endl;
64 } // main
65 /* ------------------------------------------------------------------------
66 Ausgabe des Programms Upro02:
67
68 Upro02: Jetzt geht es los!
69 Ein int-Vector der Laenge 5: 17 32 49 54 63
70 Ein int-Vector der Laenge 5: 17 32 49 54 63
71 Ein int-Vector der Laenge 5: 63 54 49 32 17
72 Upro02: Das war's erstmal!
73 ------------------------------------------------------------------------ */
```

Die beiden Funktionen `kleinerGleich` und `groesserGleich` mussten definiert werden, weil man einem Unterprogramm nur Unterprogramme mit *normalen Namen* (wie `kleinerGleich` oder

groesserGleich etc.) übergeben kann, aber ***keine Operationen*** (Funktionen mit Operatoren wie <=, >= etc. als Namen).

Aufgabe: Erweitern Sie das Programm Upro02 so, dass man damit int-Vectoren auch wie folgt sortieren kann: Zuerst kommen alle ***ungeraden*** Zahlen in ***aufsteigender*** Reihenfolge, dann alle ***geraden*** Zahlen in ***absteigender*** Folge, z. B. so (siehe v1 in Zeile 56): 17 49 63 54 32.

4.3.3 Unterprogramme mit einer variablen Anzahl von Parametern

Ein Unterprogramm, welches man mit variabel vielen Parametern aufrufen kann, bezeichnen wir hier kurz als VAP-Unterprogramm (bzw. als VAP-Funktion, VAP-Prozedur oder VAP-Methode). In Java gibt es VAP-Methoden erst seit Kurzem (ab Java 5), in C/C++ gab es solche Unterprogramme "schon immer" (d. h. seit 1970 bzw. 1980). Die bekanntesten VAP-Unterprogramme in C++ sind printf, sprintf, scanf und sscanf, in Java sind es printf und format.

In Java kann man VAP-Methoden relativ einfach und elegant programmieren. In C/C++ erfordern VAP-Unterprogramme die Verwendung von sogenannten Präprozessor-Makros (namens va_start, va_arg und va_end), die in einer Kopfdatei namens <cstdarg> definiert sind.

Beispiel01: Die VAP-Funktion add (siehe Quelldatei VarAnzParams01.cpp)

```
1   #include <cstdarg> // Ermoeglicht variable Anzahl von Parametern
2
3   long add(unsigned char anzahl, long summand1, long summand2, ...) {
4      // Diese Funktion muss mit mindestens 3 Parametern aufgerufen werden.
5      // Alle ausser dem ersten Parameter werden in diesem Beispiel als Werte
6      // des Typs long interpretiert. Der erste Parameter sollte die Anzahl
7      // der weiteren Parameter enthalten (mindestens den Wert 2).
8      long    erg = summand1 + summand2;
9      va_list summandN;                      // Typ va_list
10
11     // va_start initialisieren. Als zweiten Parameter muss man
12     // den letzten benannten ("normalen") Parameter angeben.
13     va_start(summandN, summand2);          // Makro va_start
14     for (anzahl-=2; anzahl>0; anzahl--) {
15        erg += va_arg(summandN, long);      // Makro va_arg
16     }
17     va_end(summandN);                      // Makro va_end
18
19     return erg;
20  } // add
```

Beispiel02: Die VAP-Funktion max (siehe Quelldatei VarAnzParams01.cpp)

```
1   #include <cstdarg> // Ermoeglicht variable Anzahl von Parametern
2   #include <climits> // INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, ...
3
4   int max(unsigned char anzahl, ...) {
5      // anzahl sollte die Anzahl der weiteren Parameter enthalten (mind. 0).
6      // Diese weiteren Parameter werden in diesem Beispiel als Werte des
7      // Typs int interpretiert.
8
9      int max = INT_MIN;
10     int par;                               // Wert eines Parameters
11
12     va_list parN;                          // Adresse eines Parameters
13
14     va_start(parN, anzahl);                // parN zeigt auf ersten int-Param
```

```
15     for (; anzahl>0; anzahl--) {
16
17        // Der Wert des aktuellen int-Parameters wird nach par gebracht und
18        // parN zeigt danach auf den naechsten int-Parameter:
19        par = va_arg(parN, int);
20        if (max < par) max = par;         // Maximum berechnen
21     }
22     va_end(parN);                        // Die Variable parN "aufraeumen"
23
24     return max;
25  } // max
```

4.4 Ausnahmen (exceptions)

Ausnahmen in ***Java*** haben große Ähnlichkeit mit ***Ausnahmen*** in ***C++***. Insbesondere ähneln sich die `try`-, `catch`- und `throw`-Anweisungen in beiden Sprachen sehr. Es gibt allerdings auch die folgenden Unterschiede:

Unterschied 1: In ***Java*** unterscheidet man zwischen der `throw`-Anweisung (ohne s) und einer `throws`-Klausel (mit ***s***). In C++ verwendet man in beiden Fällen das Schlüsselwort `throw` (ohne ***s***).

Unterschied 2: In C++ kann man ***Werte beliebiger Typen*** (nicht nur Objekte) als Ausnahmen werfen, also auch `int`-Werte oder `bool`-Werte etc. In Java kann man nur ***Objekte von Ausnahmeklassen*** (Unterklassen von `Throwable`) als Ausnahmen werfen.

Unterschied 3: In C++ gibt es keine `finally`-Blöcke ("die auf jeden Fall ausgeführt werden").

Unterschied 4: In C++ wird nicht zwischen ***geprüften*** und ***ungeprüften*** Ausnahmen unterschieden, sondern nur zwischen ***erwarteten*** und ***nicht-erwarteten*** Ausnahmen. Am Anfang eines C++-Unterprogramms kann man (in einer `throw`-Klausel, entspricht der Java `throws`-Klausel) Ausnahmen erwähnen, die von dem Unterprogramm evtl. ausgelöst werden. Alle in der `throw`-Klausel erwähnten Ausnahmen gelten als ***erwartet***, alle anderen als ***unerwartet***. In einem Unterprogramm ***ohne*** `throw`-Klausel gelten ***alle*** Ausnahmen als ***erwartet*** (nicht etwa als ***nicht-erwartet!***). Wenn in einem Unterprogramm eine ***unerwartete*** Ausnahme auftritt, wird eine Prozedur namens `_unexpected_handler` aufgerufen, die ihrerseits eine Prozedur namens `_terminate` aufruft die ihrerseits das Programm ***abbricht***. Mit Hilfe von Funktionen namens `set_unexpected` und `set_terminate` (Kopfdatei `<exception>`) kann man die Prozeduren `_unexpected_handler` und `_terminate` durch selbst programmierte (parameterlose) Prozeduren ersetzen.

4.4.1 throw-, try-, catch-Anweisungen (aber kein finally!)

Im folgenden Beispielprogramm werden Werte der Typen `float`, `int`, `bool` und `char` als Ausnahmen geworfen (und nicht etwa Objekte spezieller Ausnahmeklassen).

```
1   // Datei Ausnahmen01.cpp
2   /* ------------------------------------------------------------------------
3   Einfaches Beispiel fuer das Ausloesen und Behandeln von Ausnahmen.
4   In einer Schleife wird jeweils ein Zeichen eingelesen und dann eine
5   entsprechende Ausnahme ausgeloest und behandelt.
6   ---------------------------------------------------------------------- */
7   #include <iostream>
8   using namespace std;
9   // ------------------------------------------------------------------------
10  int main() {
11     cout << "Programm Ausnahmen01: Jetzt geht es los!" << endl;
12     char  eingabe;
13     float bruch1;
14     int   anzahl_b  = 0;
```

```
15      // --------------------------------------------------------------------
16      while (true) {
17         cout << "Ein Zeichen?   ";
18         cin  >> eingabe;
19         // -----------------------------------------------------------------
20         try {
21            switch (eingabe) {
22               case 'a': throw bruch1;         // ein float-Wert
23               case 'b': throw ++anzahl_b;     // ein   int-Wert
24               case 'c': throw bool();         // ein "bool-Objekt"
25               case 'q': break;                // quit
26               default : throw char(eingabe); // ein "char-Objekt"
27            } // switch
28            break;
29         } catch (float f) {cout << "Das war ein   'a'"               << endl;
30         } catch (int   i) {cout << "Das war das    " << i << ". b!"  << endl;
31         } catch (char  z) {cout << "Das war ein -->" << z            << endl;
32         } catch (...)     { // Behandler fuer alle anderen Ausnahmen.
33            cout << "Andere Ausnahme gefangen und weitergeworfen!"  << endl;
34            throw;         // Gefangene Ausnahme wird "weitergeworfen"
35         } // try/catch
36      } // while
37      // --------------------------------------------------------------------
38      // Diese Meldung erscheint nicht wenn der Benutzer ein 'c' eingibt!
39      cout << "Programm Ausnahmen01: Das war's erstmal!" << endl;
40   } // main
41   /* -----------------------------------------------------------------------
42   Ein Dialog mit dem Programm Ausnahmen01:
43
44   Programm Ausnahmen01: Jetzt geht es los!
45   Ein Zeichen?   x
46   Das war ein -->x
47   Ein Zeichen?   b
48   Das war das    1. b!
49   Ein Zeichen?   a
50   Das war ein   'a'
51   Ein Zeichen?   b
52   Das war das    2. b!
53   Ein Zeichen?   c
54   Ausnahme gefangen und weitergeworfen!
55   0 [sig] AUSNAHMEN01 1000 stackdump: Dumping stacktrace to ...
56   -------------------------------------------------------------------- */
```

4.4.2 Ein Unterprogramm mit throw-Klausel

Das folgende Programm enthält ein Beispiel für ein Unterprogramm mit einer `throw`-Klausel (siehe die Divisionsfunktion `teile` ab Zeile 19):

```
1    // Datei Ausnahmen02.cpp
2    /* -----------------------------------------------------------------------
3    Objekte einer selbst definierten Klasse (DivisonsFehler) werden als
4    Ausnahmen geworfen und gefangen.
5    -------------------------------------------------------------------- */
6    #include <iostream>
7    using namespace std;
8    // -----------------------------------------------------------------------
9    // Objekte der folgenden Klasse werden spaeter als Ausnahmen geworfen:
10   class DivisionsFehler {
11   public:
12      DivisionsFehler(int d): dividend(d) {} // Ein allgemeiner Konstruktor
13      int  getDividend() {return dividend;}  // Eine get-Funktion
```

```
14  private:
15     int dividend;
16  }; // class DivisionsFehler
17  // ------------------------------------------------------------------------
18  // Die Divisionsfunktion:
19  double teile(int dividend, int divisor) throw (DivisionsFehler) {
20     // Wandelt dividend und divisor in double-Werte um und liefert ihren
21     // Quotienten.
22     if (divisor == 0) throw DivisionsFehler(dividend);
23     return static_cast<double>(dividend) / divisor;
24  } // teile
25  // ------------------------------------------------------------------------
26  int main() {
27     int    zahl1, zahl2; // Zum Einlesen von zwei Ganzzahlen
28     double ergebnis;     // Der Quotient der beiden Ganzzahlen
29
30     while (true) {
31        cout << "Zwei Ganzzahlen ('q' zum Beenden): ";
32        // Wenn das Einlesen nicht klappt, loest cin keine Ausnahme aus,
33        // sondern liefert den Wert false (und sonst true):
34        bool eingabeHatGeklappt = cin >> zahl1 >> zahl2;
35        if (! eingabeHatGeklappt) break;   // Programm beenden
36
37        try {
38           ergebnis = teile(zahl1, zahl2); // Loest evtl. Ausnahme aus
39           cout << zahl1 << " / " << zahl2 << " ist " << ergebnis << endl;
40        } catch(DivisionsFehler ddn) {
41           cout << ddn.getDividend() << " / 0 geht nicht!"        << endl;
42        } // try/catch
43     } // while
44
45     cout << endl << "Das Programm Ausnahmen02 beendet sich!"     << endl;
46  } // main
47  /* ------------------------------------------------------------------------
48  Ein Dialog mit dem Programm Ausnahmen02:
49
50  Zwei Ganzzahlen ('q' zum Beenden): 3 12
51  3 / 12 ist 0.25
52  Zwei Ganzzahlen ('q' zum Beenden): 17 0
53  17 / 0 geht nicht!
54  Zwei Ganzzahlen ('q' zum Beenden): x
55
56  Das Programm Ausnahmen02 beendet sich!
57  --------------------------------------------------------------------- */
```

4.4.3 Unerwartete Ausnahmen selbst abfangen

Im folgenden Programm wird die Prozedur `_unexpected_handler` durch eine selbst geschriebene Prozedur (namens `NeuerUnexpectedHandler`) ersetzt.

```
1   // Datei Ausnahmen03.cpp
2   /* ------------------------------------------------------------------------
3   Die Funktion set_unexpected, eine (hoffentlich) einfache Anwendung.
4   Als "erwartet" gelten alle Ausnahmen, die in der throw-Klausel eines
5   Unterprogramms erwaehnt werden, als "unerwartet" (unexpected) alle anderen.
6   --------------------------------------------------------------------- */
7   #include <exception> // fuer set_unexpected, unexpected_handler
8   #include <string>
9   #include <iostream>
10  using namespace std;
11  // ------------------------------------------------------------------------
```

```
12 // Hilfsprozedur zum Protokollieren ("melden"):
13 void m(string s) {cout << s << endl;}
14 // ------------------------------------------------------------------------
15 // Zwei Ausnahme-Klassen:
16 class ErwarteteAusnahme   {};
17 class UnerwarteteAusnahme {};
18 // ------------------------------------------------------------------------
19 // Das Unterprogramm up1 wirft eine unerwartete Ausnahme:
20 void up1() throw (ErwarteteAusnahme) {
21    // ...
22    m("up1 wirft UnerwarteteAusnahme");
23    throw UnerwarteteAusnahme();
24 } // up1
25 // ------------------------------------------------------------------------
26 // In main wird die folgende Prozedur als "Behandler fuer unerwartete
27 // Ausnahmen" (_unexpected_handler) eingesetzt:
28 void NeuerUnexpectedHandler() {
29    m("NeuerUnexpectedHandler wirft ErwarteteAusnahme");
30    throw ErwarteteAusnahme();
31 } // NeuerUnexpectedHandler
32 // ------------------------------------------------------------------------
33 int main() {
34    // main erwartet, dass up1 eine erwartete Ausnahme wirft. Durch Ein-
35    // setzen des Unterprogramms NeuerUnexpectedHandler (als Behandler fuer
36    // unerwartete Ausnahmen, mit set_unexpected) sorgt main dafuer, dass
37    // unerwartete Ausnahmen in erwartete Ausnahmen "uebersetzt" werden:
38    cout << "Ausnahmen03: Jetzt geht es los!" << endl;
39    set_unexpected(NeuerUnexpectedHandler);
40
41    try {
42       up1();
43    } catch (ErwarteteAusnahme) {
44       m("In main ErwarteteAusnahme gefangen");
45    }
46    cout << "Ausnahmen03: Das war's erstmal!" << endl;
47 } // main
48 /* ------------------------------------------------------------------------
49 Warnung des Borland-Compilers (Version 5.5):
50
51 Warning W8078 Ausnahmen03.cpp 23: Throw expression violates exception
52 specification in function up1() throw(ErwarteteAusnahme)
53 ------------------------------------------------------------------------
54 Ausgabe des Programms Ausnahmen03:
55
56 Ausnahmen03: Jetzt geht es los!
57 up1 wirft UnerwarteteAusnahme
58 NeuerUnexpectedHandler wirft ErwarteteAusnahme
59 In main ErwarteteAusnahme gefangen
60 Ausnahmen03: Das war's erstmal!
61 --------------------------------------------------------------------- */
```

Das Beispielprogramm `Ausnahmen04` (hier nicht wiedergegeben) zeigt eine nicht-triviale Anwendung der Prozedur `set_unexpected` aus dem Buch "*The C++ Programming Language*" von *B. Stroustrup*, 3rd edition, page 378 (die Lösung wurde ins Deutsche übertragen und kommentiert).

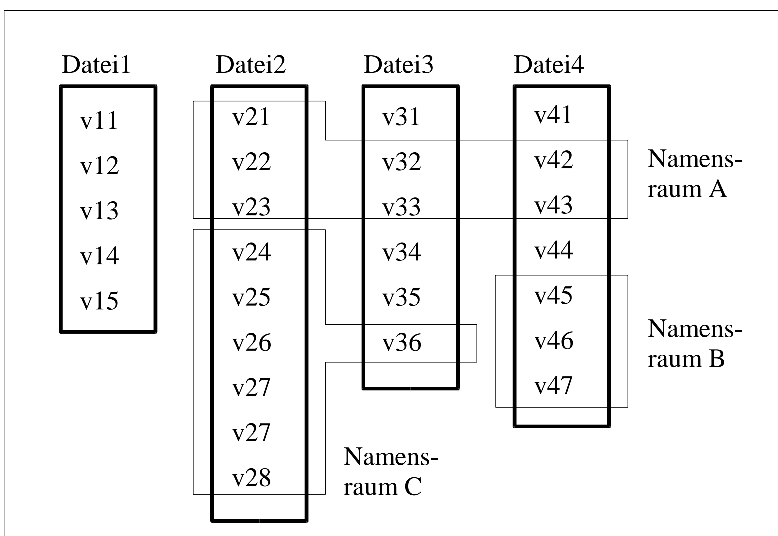
4.5 Struktur und Ausführung eines C++-Programms

Ein C++-Programm besteht aus ***Vereinbarungen*** ("Erzeugungsbefehlen"). Genau eine davon muss dem Ausführer befehlen, ein Unterprogramm namens ***main*** zu erzeugen.

Jede Vereinbarung eines C++-Programms steht in einer ***Datei*** und gehört zu einem ***Namensraum*** (namespace). In einer Datei dürfen beliebig viele Vereinbarungen stehen. Vereinbarungen, die vom Programmierer nicht ausdrücklich einem anderen Namensraum zugeordnet werden, gehören automatisch zum so genannten ***globalen Namensraum***.

Namensräume in ***C++*** haben große Ähnlichkeit mit ***Paketen*** in ***Java***. Ein Paket in Java ist ein ***Behälter*** für ***Klassen***, ***Schnittstellen*** und ***Pakete***. Ein C++-Namensraum kann ***Klassen***, ***Unterprogramme***, ***Variablen*** und ***Namensräume*** enthalten.

Dateien und Namensräume sind in folgendem Sinne ***unabhängig*** voneinander: Vereinbarungen, die in ***derselben Datei*** stehen, können zu ***verschiedenen Namensräumen*** gehören und zu ***einem Namensraum*** können Vereinbarungen aus ***verschiedenen Dateien*** gehören. Die folgende Graphik soll diese Tatsache veranschaulichen:



Dargestellt wird hier die Struktur eines Programms, welches aus ***vier Dateien*** (`Datei1` bis `Datei4`), den ***drei Namensräumen*** A bis C und dem ***globalen Namensraum*** besteht. Dabei sollen `v11`, `v12`, .., `v21`, `v22`, ... etc. die ***Vereinbarungen*** des Programms sein. Eine dieser Vereinbarungen muss ein Unterprogramm namens `main` vereinbaren.

Hier sollte man vorläufig von der (vereinfachten) Vorstellung ausgehen, dass man in ***jeder Datei*** eines Programms auf ***alle Größen*** des Programms zugreifen kann. Z. B. gehen wir erstmal davon aus, dass man ein in `Datei1` vereinbartes ***Unterprogramm*** auch in der `Datei2` ***aufrufen*** und einer in `Datei4` vereinbarten ***Variablen*** man in `Datei3` einen neuen Wert ***zuweisen*** kann. Genauere Sichtbarkeitsregeln werden später behandelt.

Wird dem Ausführer befohlen, dieses Programm auszuführen, macht er folgendes:

1. Er führt die ***Vereinbarungen*** des Programms (`v11` bis `v47`) aus, d. h. er erzeugt die dadurch vereinbarten Größen (Unterprogramme, Typen, Variablen und Konstanten). Unter anderem erzeugt er dabei das `main`-Unterprogramm.

2. Dann führt er das `main`-Unterprogramm aus. Das ist alles.

Vereinbarungen, die gemeinsam in ***einer Datei*** stehen und zum ***selben Namensraum*** gehören (z. B. `v21` bis `v23`) werden in ***der*** Reihenfolge ausgeführt, in der sie in ihrer Datei stehen. In welcher Reihenfolge Vereinbarungen ausgeführt werden, die in ***verschiedenen*** Dateien stehen bzw. zu ***verschiedenen*** Namensräumen gehören, ist ***implementierungsabhängig*** (d. h. kann von Ausführer zu Ausführer verschieden sein).

Frage: Ist der erste Satz dieses Abschnitts nicht falsch? Besteht ein C++-Programm nicht auch aus ***Anweisungen*** und ***Ausdrücken***?

Technisch gesehen ***nein***. ***Anweisungen*** und ***Ausdrücke*** dürfen ***nicht direkt*** in den Dateien des Programms stehen, sondern nur indirekt, z. B. als Teil einer Unterprogramm-Vereinbarung. Deshalb besteht das C++-Programm nur aus ***Vereinbarungen*** und nur ***die vereinbarten Größen*** (z. B. die Unterprogramme) können weitere ***Vereinbarungen***, ***Anweisungen*** und ***Ausdrücke enthalten***.

Welche Vereinbarungen zu einem C++-Programm gehören, muss bei der Übergabe des Programms an den Ausführer und vor der ersten Ausführung festgelegt werden. Das ist weniger selbstverständlich als es vielleicht klingt.

Zum Vergleich: Ein ***Java***-Programm besteht im Kern aus ***einer Hauptklasse***, die eine `main`-Methode enthalten muss. Außerdem gehören zu dem Programm alle Klassen die nötig sind, um die `main`-Methode der Hauptklasse auszuführen. Allerdings: Welche Klassen das sind, wird erst ***während der Ausführung*** des Programms entschieden und diese ***Nebenklassen*** des Programms werden nur "bei Bedarf" ***dynamisch nachgeladen***. ***Während*** ein Java-Programm schon ausgeführt wird, kann der Programmierer noch Klassen verändern, die zum Programm gehören aber erst später benötigt werden. Eine solche Flexibilität ist von der Sprache ***C++*** nicht vorgesehen.

In ***C++*** darf in den Quelldateien eines Programms nur ***ein*** `main`-Unterprogramm definiert werden. In Java darf ***jede*** Klasse eines Programms eine eigene `main`-Methode besitzen. Allerdings wird nur die `main`-Methode der ***Hauptklasse*** automatisch ausgeführt, die `main`-Methoden der Nebenklassen (falls vorhanden) verhalten sich wie gewöhnliche Klassenmethoden.

Namensräume dienen dazu, Namenskonflikte ("ein Name bezeichnet mehrere verschiedene Größen und der Ausführer weiß nicht, welche gemeint ist") zu vermeiden. Das ist vor allem dann wichtig, wenn die Dateien eines Programms von verschiedenen Programmierern oder sogar von verschiedenen Firmen erstellt werden. Hier die Grundregeln:

Angenommen, eine Größe namens `sum` wurde in einem Namensraum namens `paket1` definiert. ***Innerhalb*** des Namensraums `paket1` ist diese Größe dann direkt sichtbar und man kann man sie mit ihrem ***einfachen*** Namen `sum` bezeichnen. ***Außerhalb*** von `paket1` muss man die Größe mit dem ***zusammengesetzten*** Namen `paket1::sum` bezeichnen.

Im Wirkungsbereich einer `using`-***Direktive*** wie z. B. `using namespace std;` sind ***alle*** im Namensraum `std` definierten Größen ***direkt sichtbar***, d. h. statt `std::cout`, `std::endl`, ... etc. kann man dann einfach `cout`, `endl`, ... etc. schreiben.

Im Wirkungsbereich einer `using`-***Deklaration*** wie z. B. `using std::cout;` ist die ***eine*** Größe `cout` (definiert im Namensraum `std`) direkt sichtbar, d. h. man kann sie dann statt mit `std::cout` auch einfach mit `cout` bezeichnen.

4.6 Mehrere Quelldateien, ein Programm

In C++ unterscheidet man ***zwei Arten*** von Vereinbarungen: ***Deklarationen*** und ***Definitionen***. Eine ***Definition*** entspricht dabei dem was man in anderen Sprachen (z. B. in Java) eine ***Vereinbarung*** nennt: Eine ***C++-Definition*** ist ein Befehl (des Programmierers an den Ausführer) eine bestimmte Größe zu ***erzeugen***. Dagegen ist eine ***Deklaration*** nur ein ***Versprechen*** (des Programmierers an den Ausführer), dass die betreffende Größe in irgendeiner Datei des Programm ***definiert*** wird.

Wird eine Variable, eine Konstante oder ein Unterprogramm in einer `Datei1` *definiert* und soll in einer anderen `Datei2` *benutzt* werden, dann muss diese Größe in der `Datei2` *deklariert* werden. Hier ein (hoffentlich) einfaches Beispiel für ein Programm, welches aus mehreren (genauer: aus zwei) Quelldateien besteht und einige ***Deklarationen*** und ***Definitionen*** enthält:

```
1   // Datei Haupt.cpp
2   /* ------------------------------------------------------------------------
3   Das Programm BeispielProg17 besteht aus den beiden Quelldateien Haupt.cpp
4   und Neben.cpp. Es gibt ein paar Zeilen zur Standardausgabe aus.
5   --------------------------------------------------------------------- */
6   #include <string>
7
8   // Definition einer string-Konstanten (wird in Neben.cpp benutzt):
9   extern std::string const PROG_NAME = "BeispielProg17";
10
11  // Deklaration zweier Unterprogramme (werden in Neben.cpp definiert):
12  void gibAus   (std::string);
13  void zierZeile(char was, unsigned wieviele);
14
15  // Das main-Unterprogramm des BeispielProg17:
16  int main() {
17     zierZeile('~', 50);
18     gibAus("Jetzt geht es los!");
19     zierZeile('+', 30);
20     gibAus("Das war's erstmal!");
21     zierZeile('~', 50);
22  }
23  /* ------------------------------------------------------------------------
24  Befehl zum Compilieren und Binden dieses Programms mit dem Gnu-Compiler:
25
26  > g++ -o BeispielProg17.exe Neben.cpp Haupt.cpp
27  ---------------------------------------------------------------------------
28  Befehl zum Compilieren und Binden dieses Programms mit dem Borland-Compiler:
29
30  > bcc32 -eBeispielProg17.exe Neben.cpp Haupt.cpp
31  --------------------------------------------------------------------- */
```

```
1   // Datei Neben.cpp
2   /* ------------------------------------------------------------------------
3   Das Programm BeispielProg17 besteht aus den beiden Quelldateien Haupt.cpp
4   und Neben.cpp. Es gibt ein paar Zeilen zur Standardausgabe aus.
5   --------------------------------------------------------------------- */
6   #include <iostream>
7   #include <string>
8
9   // Deklaration einer string-Konstanten (wird in Haupt.cpp definiert)
10  extern std::string const PROG_NAME;
11
12  // Definition zweier Unterprogramme (werden in Haupt.cpp benutzt)
13  void gibAus(std::string s) {
14     std::cout << PROG_NAME << ": " << s << std::endl;
15  }
```

```
16
17  void zierZeile(char c, unsigned n) { // Andere Param.namen als in Haupt.cpp
18     std::string z(n, c); // Der String z enthaelt n mal das Zeichen c
19     gibAus(z);;
20  }
21  /* ------------------------------------------------------------------------
22  Ausgabe des Programms BeispielProg17:
23
24  BeispielProg17: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
25  BeispielProg17: Jetzt geht es los!
26  BeispielProg17: ++++++++++++++++++++++++++++++
27  BeispielProg17: Das war's erstmal!
28  BeispielProg17: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
29  ------------------------------------------------------------------------ */
```

Aufgabe: Tragen Sie die richtigen Dateinamen in die folgende Tabelle ein:

Größe	***wird definiert in Datei***	***wird benutzt in Datei(en)***
PROG_NAME		
gibAus		
zierZeile		
main		

4.7 Compilieren und Binden, Deklarationen und Definitionen in Dateien

Die Sprache ***Java*** beruht auf einem relativ ***modernen*** Compilationsmodell und ein Java-Programmierer braucht sich in aller Regel nicht im Einzelnen darum zu kümmern, welche Aufgaben vom Java -Compiler und welche vom Java-Interpreter (der virtuellen Maschine) erledigt werden. Es genügt, wenn er sich einen ***abstrakten Java-Ausführer*** vorstellt, dem er seine Programme übergibt und der diese Programme prüft und ausführt.

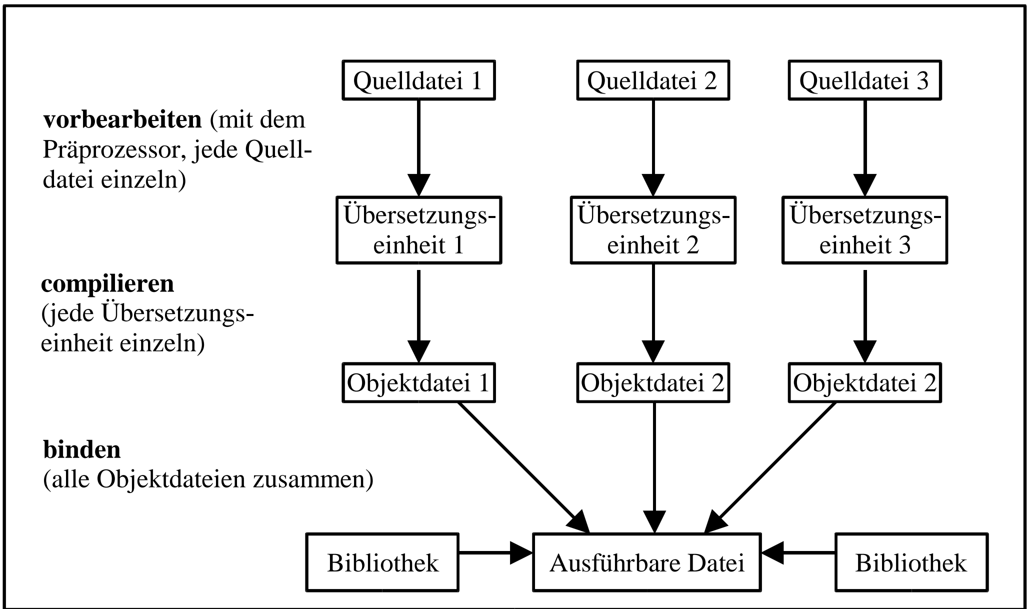
Die Sprache ***C++*** beruht auf einem ***relativ alten*** (von der Sprache C übernommenen) Compilationsmodell und man kann zahlreiche Regeln der Sprache nur dann richtig verstehen, wenn man zumindest mit den Grundzügen dieses Modells vertraut ist.

Ein C++-Ausführer besteht unter anderem aus einem ***Compiler*** und aus einem ***Binder***. Der Compiler prüft jeweils ***eine Quelldatei*** und übersetzt sie (falls er keine formalen Fehler findet) in eine entsprechende ***Objektdatei***. Der Binder prüft alle Objektdateien eines Programms noch einmal und bindet sie (falls er keine formalen Fehler findet) zu einer ***ausführbaren Datei*** zusammen.

Anmerkung 1: Der Begriff einer ***Objektdatei*** ist älter als die objektorientierte Programmierung und hat ***nichts*** mit ihr zu tun. Eine Objektdatei enthält keine Objekte (im Sinne der OO-Programmierung), sondern das Ergebnis der Übersetzung ***einer*** Quelldatei. Objektdateien haben häufig Namen mit Erweiterungen wie `.o` oder `.obj` (je nach verwendetem Compiler).

Anmerkung 2: Unter DOS und Windows bezeichnet man die ***ausführbaren Dateien***, die ein C++-Binder erzeugt, auch als ***.exe-Dateien***. Unter einem Unix-Betriebssystem erkennt man ausführbare Dateien nicht an ihrem Namen, sondern an bestimmten ***Ausführungsberechtigungsbits***. Man kann aber auch unter Unix ausführbaren Dateien Namen mit der Erweiterung `.exe` geben. Dadurch erleichtert man sich z. B. das Löschen aller ausführbaren Dateien in einem bestimmten Verzeichnis.

Die folgende Graphik soll anschaulich machen, wie aus mehreren ***Quelldateien*** eines Programms eine ***ausführbare Datei*** erzeugt wird:



Der Präprozessor (preprocessor) erweitert jede ***Quelldatei*** (each source file) zu einer ***Übersetzungseinheit*** (translation unit), indem er alle Präprozessor-Befehle in der Quelldatei ausführt (er erkennt "seine" Befehle daran, dass sie mit einem Nummernzeichen # beginnen). Z. B. ersetzt der Präprozessor jeden `#include`-Befehl durch den Inhalt der entsprechenden Datei. Der Präprozessor wird meistens automatisch vom Compiler aufgerufen.

Der Compiler übersetzt jede ***Übersetzungseinheit*** in eine ***Objektdatei*** (object file). Der Binder (the linker) bindet alle Objektdateien zu einer ausführbaren Datei (unter Windows: zu einer `.exe`-Datei) zusammen und fügt dabei, falls nötig, noch fehlende Definitionen aus seinen Bibliotheken ein.

Häufig betrachtet man den Präprozessor und den eigentlichen Compiler als ***ein*** Werkzeug, welches Quelldateien in Objektdateien übersetzt, und bezeichnet dieses Kombiwerkzeug als Compiler.

Das ***Grundproblem*** dieses Compilationsmodells: Der Compiler prüft und übersetzt immer nur **eine** Quelldateiatei auf einmal und "hat kein Gedächtnis". Das heisst: Wenn er eine Datei übersetzt, weiss er nichts von irgendwelchen anderen Dateien, die möglicherweise zum selben Programm gehören. Wird eine Größe z. B. in der Quelldatei 1 ***definiert*** und in der Quelldatei 2 ***benutz***, dann prüft der Compiler ***nicht***, ob die Definition und die Benutzung "zusammenpassen". Erst der Binder prüft, ob alle ***Objektdateien*** eines Programms konsistent sind ("zusammenpassen") und ob jede Größe, die irgendwo ***benutzt*** wird, auch irgendwo ***definiert*** wird (oder ob sich eine geeignete Definition in einer Bibliothek befindet).

Damit trotzdem schon der ***Compiler*** ein paar wichtige Prüfungen durchführen kann, hat man in C++ zusätzlich zu den ***eigentlichen*** Vereinbarungsbefehlen (den ***Definitionen***) die ***Deklarationen*** eingeführt, die dem Compiler ***versprechen***, dass eine bestimmte Größe in irgendeiner Datei des Programms definiert wird. Der Compiler ist "gutgläubig" und vertraut diesen Versprechungen, ohne sie zu überprüfen. Erst der Binder ist ein bisschen misstrauischer und führt Konsistenzprüfungen durch. Es folgen die wichtigsten Regeln zu diesem Thema.

4.8 Regeln über Deklarationen und Definitionen (in Dateien)

Zur Erinnerung: Eine ***Definition*** ist ein Befehl (des Programmierers an den Ausführer), eine Größe zu erzeugen. Eine ***Deklaration*** ist ein Versprechen (des Programmierers an den Ausführer), eine Größe in irgendeiner Datei eines Programms zu definieren.

DekDef 1: Man darf ***jede Größe*** in ***jeder Datei*** eines Programms ***beliebig oft deklarieren*** (d. h. man darf beliebig oft versprechen, die Größe zu definieren).

DekDef 2: Eine ***UV-Größe*** (d. h. ein ***Unterprogramm***, eine ***Variable*** oder eine ***Konstante***) darf man in höchstens ***einer*** Datei eines Programms (einmal) ***definieren***. Diese Regel ist auch unter dem Namen ***one definition rule*** (ODR) bekannt. Ein Versprechen, eine UV-Größe zu definieren, muss man nur dann einhalten, wenn die Größe irgendwo im Programm ***benutzt*** wird.

DekDef 3: Will man eine UV-Größe `x` in einer Datei `d` benutzen, muss man `x` in `d` mindestens ***deklarieren*** (oder sogar definieren), und zwar ***vor*** der ersten Benutzung von `x`.

DekDef 4: Einen ***Typ*** (der nicht vordefiniert ist wie `int` oder `bool`) muss man in jeder Datei ***definieren***, in der er ***benutzt*** wird (und zwar ***vor*** der ersten Benutzung). Die ODR gilt also ***nicht*** für Typen, sondern nur für UV-Größen!

DekDef5: Mit einem `typedef`-Befehl kann man einem (schon existierenden) Typ einen ***zusätzlichen Namen*** (einen Alias-Namen) geben, z. B. so:

```
typedef unsigned long ulong;
```

Nach Ausführung dieses Befehls hat der Typ `unsigned long` zusätzlich auch den Namen `ulong` (und für den Compiler macht es keinen Unterschied, mit welchem der Namen man den Typ bezeichnet). Ein solcher `typedef`-Befehl gilt als ***Deklaration*** und kann in jeder Datei eines Programms ***beliebig oft*** wiederholt werden. Die Silbe `def` am Ende von `typedef` wurde nur eingeführt, um leichtgläubige Menschen zu verwirren und vom Programmieren in C++ abzuschrecken.

Das folgende Programm soll die Regeln zu Deklarationen und Definitionen durch Beispiele verdeutlichen. Alle Deklarationen stehen (unmittelbar nacheinander) zweimal da, um die Regel **DekDef 1** (siehe oben) zu veranschaulichen.

```
1   // Datei DekDef01.cpp
2   /* ------------------------------------------------------------------------
3   Deklarationen und Definitionen (eine fast vollstaendige Sammlung von
4   Beispielen).
5   ------------------------------------------------------------------------ */
6   #include <iostream>
7   using namespace std;
8   // ========================================================================
9   // DEKLARATIONEN von Variablen, Konstanten und Unterprogrammen:
10
11  extern       int n1;            // Dekl. einer Variablen,      extern muss
12  extern       int n1;            // Dekl. einer Variablen.      extern muss
13  extern const int k1;            // Dekl. einer Konstanten,     extern muss
14  extern const int k1;            // Dekl. einer Konstanten,     extern muss
15  extern       int plus1 (int  ); // Dekl. eines Unterprogramms, extern kann
16  extern       int plus1 (int n); // Dekl. eines Unterprogramms, extern kann
17               int minus1(int  ); // Dekl. eines Unterprogramms, extern kann
18               int minus1(int n); // Dekl. eines Unterprogramms, extern kann
19  // Die Deklaration eines Unterprogramms kann, muss aber nicht, mit
20  // dem Schluesselwort "extern" gekennzeichnet werden. Am fehlenden
21  // Rumpf erkennt der Ausfuehrer, dass es sich um eine Deklaration
22  // und nicht um eine Definition handelt.
23  // ------------------------------------------
24  // DEKLARATIONEN von Typen (static und extern duerfen *nicht*).
```

```
25 // Ob man einen Typ mit "class" oder "struct" *deklariert* ist egal:
26 struct meinStrucTyp1;
27 struct meinStrucTyp1;
28 class  meinClassTyp1;
29 class  meinClassTyp1;
30 struct meinClassTyp1;  // Als class  oder struct deklariert? Egal!
31 class  meinStrucTyp1;  // Als struct oder  class deklariert? Egal!
32 enum   meinEnumTyp1;
33 enum   meinEnumTyp1;
34 // ---------------------------------------------
35 // Der Gnu-Cygnus-C++-Compiler (Version 2.95.2) erlaubt noch keine
36 // *Deklarationen* von enum-Typen, nur *Definitionen* von enum-Typen.
37 // ---------------------------------------------
38 // DEKLARATIONEN von zusaetzlichen Typnamen fuer schon vorhandene Typen:
39 typedef int ganz;       // Der Typ int heisst jetzt auch noch ganz
40 typedef int ganz;
41 typedef int integer;    // Der Typ int heisst jetzt auch noch integer
42 typedef int integer;
43 typedef meinEnumTyp1 * adresseVon MeinEnumTyp1;
44 typedef meinEnumTyp1 * adresseVon MeinEnumTyp1;
45 // ==================================================================
46 // DEFINITIONEN von Variablen, Konstanten und Unterprogrammen:
47              int n1;
48              int n2 = 17;
49 static       int n3;
50 static       int n4 = 25;
51
52        const int k1 =  3;
53 extern const int k2 = 13;
54
55              int plus1 (int g) {return g+1;}
56 static       int minus1(int g) {return g-1;}
57 // ---------------------------------------------
58 // DEFINITIONEN von Typen.
59 // Ob man einen Typ mit "class" oder "struct" *definiert* ist *fast* egal:
60 // Bei struct-Typen ist public  Standard,
61 // bei  class-Typen ist private Standard,
62 // -----------------------
63 struct meinStrucTyp1 {
64    meinStrucTyp1(int n) {wert = n;}    // Ein  oeffentlicher Konstruktor
65    int getWert()        {return wert;} // Eine oeffentliche Methode
66 private:
67    int wert;                           // Ein  privates Attribut
68 }; // struct meinStrucTyp1
69 // -----------------------
70 class  meinClassTyp1  {
71    int wert;                           // Ein  privates Attribut
72 public:
73    meinClassTyp1(int n) {wert = n;}    // Ein  oeffentlicher Konstruktor
74    int getWert()        {return wert;} // Eine oeffentliche Methode
75 }; // meinClassTyp1
76 // -----------------------
77 enum meinEnumTyp1 {rot, gruen, blau};  // Ein Aufzaehlungs-Typ (enum-type)
78 // ==================================================================
79 // Fehlerhafte Vereinbarungen:
80          extern int n5;
81 //static        int n5;
82
83 //static struct meinStrucTyp2;
84 //static class  meinClassTyp2;
85 //extern struct meinStrucTyp3;
```

```
86  //extern class  meinClassTyp3;
87  // ======================================================================
88  int main() {
89     // Jetzt werden alle definierten Groessen (Variablen, Konstanten,
90     // Unterprogramme und Typen) und die deklarierten typedef-Namen auch mal
91     // benutzt:
92
93     cout << "DekDef01: Jetzt geht es los!"  << endl;
94
95     ganz                     g1 =  5;
96     integer                  g2 = 15;
97     adresseVon MeinEnumTyp1  z  = new meinEnumTyp1(blau);
98
99     cout << "g1: " << g1  << ", g2: " << g2 << endl;
100    cout << "*z: " << *z                    << endl;
101
102    cout << "n1: " << n1  << ", n2: " << n2 << endl;
103    cout << "n3: " << n3  << ", n4: " << n4 << endl;
104    cout << "k1: " << k1  << ", k2: " << k2 << endl;
105
106    cout << "plus1 (17)           : " << plus1(17)           << endl;
107    cout << "minus1(17)           : " << minus1(17)          << endl;
108
109    meinStrucTyp1 struct01(77);
110    meinClassTyp1 objekt01(88);
111    cout << "struct01.getWert()   : " << struct01.getWert()  << endl;
112    cout << "objekt01.getWert()   : " << objekt01.getWert()  << endl;
113
114    meinEnumTyp1 meineLieblingsFarbe = gruen;
115    cout << "meineLieblingsFarbe  : " << meineLieblingsFarbe << endl;
116
117    cout << "DekDef01: Das war's erstmal!"  << endl;
118 } // main
119 /* ----------------------------------------------------------------------
120 Fehlermeldungen des Gnu-Cygnus-Compilers (Version 2.95.2), wenn die
121 entsprechenden Zeilen keine Kommentare sind:
122
123 DekDef01.cpp:81: `n5' was declared `extern' and later `static'
124 DekDef01.cpp:80: previous declaration of `n5'
125 DekDef01.cpp:83: `static' can only be specified for objects and functions
126 DekDef01.cpp:84: `static' can only be specified for objects and functions
127 DekDef01.cpp:85: `extern' can only be specified for objects and functions
128 DekDef01.cpp:86: `extern' can only be specified for objects and functions
129 ----------------------------------------------------------------------
130 Fehlermeldungen des Borland C++-Compilers (Version 5.5.1 for Win32):
131
132 D:\meineDateien\BspC++\Strukt\DekDef01.cpp:
133 Error E2092 ...DekDef01.cpp 83: Storage class 'static' is not allowed here
134 Error E2092 ...DekDef01.cpp 84: Storage class 'static' is not allowed here
135 Error E2092 ...DekDef01.cpp 85: Storage class 'extern' is not allowed here
136 Error E2092 ...DekDef01.cpp 86: Storage class 'extern' is not allowed here
137 ----------------------------------------------------------------------
138 Ausgabe des Programms DekDef01:
139
140 DekDef01: Jetzt geht es los!
141 g1: 5, g2: 15
142 *z: 2
143 n1: 0, n2: 17
144 n3: 0, n4: 25
145 k1: 3, k2: 13
146 plus1 (17)        : 18
147 minus1(17)        : 16
```

```
148 struct01.getWert() : 77
149 objekt01.getWert() : 88
150 meineLieblingsFarbe: 1
151 DekDef01: Das war's erstmal!
152 ------------------------------------------------------------------------ */
```

Wenn man in einem Programm gewisse Größen (z. B. eine Variable `sum` und eine Funktion `plus1`) in ***mehreren*** Dateien ***benutzen*** will, kann man die ***Definitionen*** und ***Deklarationen*** dieser Größen im Prinzip wie folgt anordnen:

```
// Datei dat1.cpp
// Definitionen einiger
// Groessen:

int sum = 35;
int plus1(int n) {
   return n+1;
}
...
```

```
// Datei dat2.cpp
// Deklarationen der
// Groessen:

extern int sum;
int plus1(int);

// Benutzung der
// Groessen:
...
sum = sum + 3;
...
sum = plus1(sum);
...
```

```
// Datei dat3.cpp
// Deklarationen der
// Groessen:

extern int sum;
int plus1(int);

// Benutzung der
// Groessen:
...
sum = plus1(sum);
...
if (sum > 100) ...;
...
```

In ***jeder*** Datei, in der man die Größen benutzen will, muss man sie ***deklarieren***. Wenn man die Deklarationen "von Hand" in jede Datei schreibt, können sich leicht Schreibfehler einschleichen. Und wenn Änderungen notwendig werden, müsste man sie an ***jeder Kopie*** der Deklarationen vornehmen. Auch das ist fehlerträchtig. In aller Regel geht man deshalb wie folgt vor:

Man schreibt die ***Deklarationen*** in eine so genannte ***Kopfdatei*** (header file) und inkludiert diese (mit dem `#include`-Befehl) in ***jeder*** Datei, in der die Größen ***benutzt*** werden. Außerdem inkludiert man die Kopfdatei auch in der Datei, in der man die Größen definiert (warum? Siehe unten). Der `include`-Befehl wird ***unmittelbar vor*** dem eigentlichen Compilieren automatisch durch den Inhalt der inkludierten Kopfdatei ersetzt (allerdings nur "vorübergehend" und nicht permanent, ***nach*** dem Compilieren haben alle ***Quell***- und ***Kopfdateien*** noch ***denselben*** Inhalt wie vorher). Hier ein Beispiel, wie Deklarationen und Definitionen auf verschiedene Dateien verteilt werden können:

```
// Kopfdatei dat1.h
// Deklarationen der
// Groessen:

extern int sum;
int plus1(int);
```

```
// Datei dat1.cpp
#include "dat1.h"

// Definitionen der
// Groessen:

int sum = 35;
int plus1(int n) {
   return n+1;
}
...
```

```
// Datei dat2.cpp
#include "dat1.h"

// Benutzung der
// Groessen:
...
sum = sum + 3;
...
sum = plus1(sum);
...
```

```
// Datei dat3.cpp
#include "dat1.h"

// Benutzung der
// Groessen:
...
sum = plus1(sum);
...
if (sum > 100) ...
...
```

Das "Männchen" links in diesem Diagramm besteht aus der ***Kopfdatei*** namens `dat1.h` und der zugehörigen ***Definitionsdatei*** (oder: ***Implementierungsdatei***) `dat1.cpp`.

Aufgabe: Warum inkludiert man die Kopfdatei `dat1.h` auch in der Implementierungsdatei `dat1.cpp`, selbst wenn die Größen (im Beispiel: `sum` und `plus1`) dort ***nicht benutzt*** werden? Was für Fehler kann der Ausführer nur dann automatisch entdecken, wenn man das tut?

Merke: ***Kopfdateien*** (header files) werden (für sich allein) normalerweise ***nicht compiliert***, sondern nur in andere Dateien ***inkludiert*** (und dann als Teil dieser anderen Dateien compiliert). Es gibt allerdings auch Compiler, mit denen man häufig benutzte und/oder besonders umfangreiche Kopfdateien für sich allein compilieren kann, wodurch sie in ein kompakteres, für den Compiler schneller bearbeitbares Format überführt werden.

Kopfdateien sind also nur deshalb notwendig, weil ein ***C++-Compiler*** immer nur ***eine*** Datei auf einmal compiliert und sich dabei keine anderen Dateien "ansieht". In ***Java*** braucht der Programmierer keine Kopfdateien zu schreiben und zu verwalten, weil der Compiler beim compilieren ***einer*** Datei automatisch auch auf ***andere*** Quelldateien zugreift und das Zusammenpassen der Dateien prüft. Kurz: ***Java***-Compiler ***haben ein Gedächtnis*** und benötigen ***keine*** Kopfdateien, ***C++***Compiler haben ***kein*** Gedächtnis und deshalb "erinnert" man sie mit Hilfe von Kopfdateien.

Weitere Regeln über Deklarationen, Definitionen und Kopfdateien werden später behandelt.

4.9 Daten mit printf formatieren und ausgeben

Wenn man Daten zum Bildschirm oder in eine Textdatei ausgibt, muss man sie nicht nur in Strings umwandeln, sondern häufig auch noch weitergehend ***formatieren***. Damit Zahlen übersichtliche Spalten bilden, muss man sie unabhängig von ihrer Größe in einer bestimmten Breite ausgeben, ein positives Vorzeichen soll manchmal ausgegeben und manchmal unterdrückt werden. Bruchzahlen sollen nur mit einer bestimmten Anzahl von Nachpunktstellen erscheinen und müssen vorher eventuell gerundet werden. Texte sollen manchmal linksbündig und manchmal rechtsbündig ausgegeben werden etc.

Mit dem Unterprogramm `printf` kann man Daten unterschiedlicher Typen formatieren und zur Standardausgabe ausgeben, etwa so wie im folgenden Beispielprogramm `printf01.cpp`:

```
1   // Datei printf01.cpp
2
3   #include <cstdio> // E/A in Bytestroeme, printf, scanf, sprintf, ...
4                     // fopen, fclose, fflush, fgets, fputs, fread, fwrite, ...
5   using namespace std;
6   // ---------------------------------------------------------------------
7   int main() {
8      printf("printf01: Jetzt geht es los!\n");
9
10     // Ein paar Variablen gaengiger Type:
11     int    i01  = 123456;
12     int    i02  = 23;
13     int    i03  = 50;
14     double d01  = 123.45678;
15
16     // printf-Befehle mit einem einzigen Parameter:
17     printf("--------------------------------------\n");
18     printf("A Whiskey enthaelt 54%% Alkohol!\n");
19
20     // printf-Befehle mit zwei Parametern, Ganzzahlen:
21     printf("--------------------------------------\n");
22     printf("B Der Preis betraegt %d Euro.\n",    23);
23     printf("C Der Preis betraegt %d Euro.\n",  i01);
24     printf("D Der Preis betraegt %7d Euro.\n", i02);
25     printf("E Der Preis betraegt %7d Euro.\n", i01);
26     printf("F Der Preis betraegt %7d Euro.\n", 123456789);
27
28     // printf-Befehle mit mehr Parametern, Ganzzahlen:
29     printf("--------------------------------------\n");
30     printf("G Das kostet %d Euro und %d Cents.\n", i02, i03);
31     printf("H 3 Zahlen: %d, %d, %f.\n",            i02, i03, d01);
32
33     // printf-Befehle mit zwei Parametern, Bruchzahlen:
34     printf("--------------------------------------\n");
35     printf("I Ein Euro kostet %f US-Cents.\n",      d01);
36     printf("J Ein Euro kostet %10.4f US-Cents.\n",  d01);
37     printf("K Ein Euro kostet %10.2f US-Cents.\n",  d01);
38     printf("L Ein Euro kostet %-10.4f US-Cents.\n", d01);
39     printf("M Ein Euro kostet %-10.2f US-Cents.\n", d01);
40     printf("N Ein Euro kostet %3.1f US-Cents.\n",   d01);
41     printf("--------------------------------------\n");
42     printf("printf01: Das war's erstmal!\n");
43  } // main
```

Das Unterprogramm `printf` darf man mit einem oder mehreren Parametern aufrufen. Der erste Parameter sollte ein String sein (ein "alter C-String", z. B. ein String-Literal wie `"Summe: %d\n"`, nicht

ein Objekt der Klasse `string`!) und wird auch als ***der Formatstring*** des `printf`-Befehls bezeichnet. Dieser Formatstring darf beliebigen Text und darin (0 oder mehr) ***Umwandlungsbefehle*** enthalten.

Ein Umwandlungsbefehl beginnt mit einem Prozentzeichen `%` und endet mit einem speziellen Umwandlungsbuchstaben wie `d`, `f` oder `x` etc. (`d` wie "Dezimalzahl", `f` wie "Float-Zahl", `x` wie "Hex-Zahl"). In Zeile 30 wird `printf` mit einem Formatstring aufgerufen, der ***zwei*** Umwandlungsbefehle enthält (beide gleich `%d`). Mit einem Umwandlungsbefehl `%d` befiehlt man dem Ausführer, einen `int`-Wert in eine Dezimalzahl umzuwandeln.

Zwischen dem Prozentzeichen und dem Umwandlungsbuchstaben können weitere Zeichen stehen, die die gewünschte Umwandlung (oder: Formatierung) näher beschreiben. In Zeile 38 steht in Formatstring des `printf`-Befehls der Umwandlungsbefehl `%-10.4f`. Er befiehlt dem Ausführer, einen `float-` oder `double`-Wert in einen Dezimalbruch umzuwandeln, der mit einem Vorzeichen + oder – beginnt, ingesamt mindestens 10 Zeichen lang ist und 4 Nachpunktstellen hat.

Ein doppeltes Prozentzeichen `%%` in einem Formatstring ist kein normaler Umwandlungsbefehl, sondern steht einfach für ein einfaches Prozentzeichen `%` (siehe oben Zeile 18 und unten die entsprechende Ausgabe in Zeile 46).

Für jeden Umwandlungsbefehl im Formatstring eines `printf`-Befehls muss man (normalerweise) nach dem Formatstring noch einen ***weiteren Parameter*** angeben, dessen Typ zum Umwandlungsbefehl passen sollte (zu `%d` passt `int`, zu `%f` passen `float` und `double` etc.). Wenn man zuviele weitere Parameter angibt, werden die letzten davon einfach ignoriert. Wenn man zuwenig weitere Parameter angibt, akzeptiert der Ausführer (d. h. der Compiler) das Programm, aber bei der Ausführung treten dann in aller Regel merkwürdige Fehler auf. Normalerweise entspricht jedem Umwandlungsbefehl im Formatstring ein weiterer Parameter.

Ein `printf`-Befehl bewirkt, dass sein Formatstring ausgegeben wird, nachdem darin jeder Umwandlungsbefehl durch den entsprechend formatierten weiteren Paramter ersetzt wurde.

Die Ausgabe des Programms `printf01` sieht etwa so aus:

```
44  printf01: Jetzt geht es los!
45  ------------------------------------
46  A Whiskey enthaelt 54% Alkohol!
47  ------------------------------------
48  B Der Preis betraegt 23 Euro.
49  C Der Preis betraegt 123456 Euro.
50  D Der Preis betraegt      23 Euro.
51  E Der Preis betraegt  123456 Euro.
52  F Der Preis betraegt 123456789 Euro.
53  ------------------------------------
54  G Das kostet 23 Euro und 50 Cents.
55  H 3 Zahlen: 23, 50, 123.456780.
56  ------------------------------------
57  I Ein Euro kostet 123.456780 US-Cents.
58  J Ein Euro kostet   123.4568 US-Cents.
59  K Ein Euro kostet     123.46 US-Cents.
60  L Ein Euro kostet 123.4568   US-Cents.
61  M Ein Euro kostet 123.46     US-Cents.
62  N Ein Euro kostet 123.5 US-Cents.
63  ------------------------------------
64  printf01: Das war's erstmal!
65  ---------------------------------------------------------------------- */
```

Das Unterprogramm `printf` ist wohl der vielseitigste und komplizierteste Befehl der Sprachen C und C++. Er ist sehr mächtig, bietet aber auch viele Möglichkeiten, Fehler zu machen, die weder vom Compiler noch vom Binder entdeckt werden (siehe dazu das Beispielprogramm `printf02.cpp`,

hier nicht wiedergegeben). Leider bringen viele dieser Fehler ein Programm zwar zum Absturz (gut!), bewirken aber nicht, dass eine verständliche Fehlermeldung ausgegeben wird (schlecht).

Ein weiterer wichtiger Kritikpunkt: Der Programmierer kann den `printf`-Befehl nicht erweitern, um damit Werte seiner selbst vereinbarten (Klassen- und `enum`-) Typen zu formatieren und auszugeben. David Hanson schlägt in seinem Buch "C Interfaces and Implementations" vor, den `printf`-Befehl durch einen selbst programmierten und ähnlichen, aber systematischeren und erweiterbaren Befehl zu ersetzen.

Java 5 enthält ebenfalls einen `printf`-Befehl, der dem gleichnamigen C/C++-Befehl stark nachempfunden ist. Dieser Java-Befehl ist aber erweiterbar und deutlich leichter zu lernen und anzuwenden als der entsprechende C/C++-Befehl (weil Java ein stärkeres Typenkonzept hat als C++, und weil Java-Programme nicht unkontrolliert abstürzen, sondern nur normal oder durch eine Ausnahme mit einer Fehlermeldung beendet werden).

Das C++-Unterprogramm `sprintf` funktioniert ganz entsprechend wie `printf`, gibt die formatierten Daten aber nicht zur Standardausgabe aus, sondern schreibt sie in einen (alten C-) String.

Mit dem Unterprogramm `scanf` kann man Textdaten von der Standardeingabe einlesen, in interne Werte umwandeln und in entsprechenden Variablen abspeichern lassen. Das Unterprogramm `sscanf` funktioniert ganz entsprechend, liest aber aus einem (alten C-) String.

4.10 Was bedeutet das Schlüsselwort static?

In C++ ist das Schlüsselwort `static` mit mindestens drei verschiedenen Bedeutungen überladen (genauer: ***überlastet***).

1. Wenn man eine Variable direkt in einer Quelldatei vereinbart (und nicht in einem Unterprogramm oder in einer Klasse etc.), dann kann man sie mit `static` kennzeichnen, um sie zu schützen. Die Variable ist dann nur in dieser einen Quelldatei (ihrer Heimatdatei) sichtbar und in allen anderen Dateien des selben Programms nicht sichtbar. Ganz entsprechendes gilt auch für Unterprogramme (siehe dazu den Abschnitt ***14.1 Der Stapel-Modul StapelM***).

2. Wenn man eine Klasse definiert, muss man (ganz ähnlich wie in Java) die Klassenelemente mit `static` kennzeichnen, um sie von Objektelementen zu unterscheiden (siehe dazu den Abschnitt ***12 Klassen als Module und Baupläne für Module***).

3. Wenn man eine Variable V in einem Unterprogramm U vereinbart und mit `static` kennzeichnet, dann wird V bei der ersten Ausführung von U erzeugt, aber nicht zerstört, wenn U fertig ausgeführt ist. Vielmehr lebt die Variable V dann, bis das gesamte umgebende Programm fertig ausgeführt ist. Bei jeder Ausführung von U hat die Variable V den Wert, den die vorige Ausführung darin zurückgelassen hat, wie im folgenden Beispiel:

Beispiel-01: Ein Unterprogramm mit einer `static`-Variablen

```
1   // Datei Static01.cpp
2   ...
3   char einZeichen() {
4      static char zeichen = 'A' - 1;
5      zeichen++;
6      return zeichen;
7   } // einZeichen
8   // ------------------------------------------------
9   int main() {
10     for (int i=0; i<28; i++) cout << einZeichen();
11     cout << endl;
12  } // main
```

Die Ausgabe dieses Programms sieht etwa so aus:

```
13  ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
```

Die Variable `zeichen` ist nur innerhalb des Unterprogramms `einZeichen` sichtbar, aber sie lebt so lange, wie eine außerhalb aller Unterprogramme (direkt in einer Quelldatei des Programms)vereinbarte Variable. Solche `static`-Variablen bilden eine Art ***privates Gedächtnis***, in dem das Unterprogramm sich Informationen über mehrere Ausführungen hinweg merken kann (während die nicht-static-Variablen nach jeder Ausführung des Unterprogramms zerstört werden).

5 Übersicht über alle Arten von C++-Typen (und ein Vergleich mit Java)

Das ***Typensystem*** von ***Java*** ist relativ einfach und leicht überschaubar. Hier eine Übersicht über alle ***Arten von Typen***, die man in Java unterscheidet. Die ***Einrückungen*** sollen deutlich machen, welche Art oder Gruppe von Typen in welcher anderen Art oder Gruppe enthalten ist:

- Alle Java-Typen
  - Primitive Typen
    - `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
  - Referenztypen
    - Reihungstypen (array types)
      - z. B. `int[]`, `String[][]`...
    - Klassen (-typen, class types)
      - z. B. `String`, `Vector`, ...
    - Schnittstellen (-typen, interface types)
      - z. B. `Runnable`, `Serializable`, ...

Eine entsprechende Übersicht über alle Arten von ***C++-Typen*** ist deutlich ***umfangreicher*** und ***kompli-zierter***. Im ***C++-Standard*** werden folgende Arten von Typen unterschieden:

- Alle C++-Typen
  - Fundamentale Typen (fundamental types)
    - Arithmetische Typen (arithmetic types)
      - Ganzzahltypen (integral types or integer types)
        - Vorzeichenbehaftete Ganzzahltypen (signed integral types)
          - `signed char`, `short int`, `int`, `long int`
        - Vorzeichenlose Ganzzahltypen (unsigened integral types)
          - `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`
        - Sonstige Ganzzahltypen
          - `bool`, `char`, `wchar_t`
      - Gleitpunkttypen (floating point types)
        - `float`, `double`, `long double`
    - Der leere Typ
      - `void`
  - Zusammengesetzte Typen (compound types)
    - Reihungstypen (array types)
      - z. B. `int[]`, `string[][]`, ..
    - Unterprogrammtypen (function types)
      - z. B. `bool (*) (int, int)`, ...
    - Adresstypen (pointer types)
      - z. B. `int*`, `string*`, ...
    - Referenztypen (reference types)
      - z. B. `int&`, `string&`, ...
    - Klassentypen (class types)
      - z. B. `string`, `vector<int>`, ...
    - Vereinigungstypen (union types)
    - Aufzählungstypen (enumeration types)
    - Elementadresstypen (types of pointers to non-static class members)

Diese Unterscheidung von Typ-Arten ist häufig sehr nützlich (um z. B. zu verstehen, dass die meisten Sprachregeln für ***Ganzzahltypen*** auch auf die Typen `char` und `bool` zutreffen). Manchmal ist es allerdings günstiger, zwischen folgenden Arten von C++-Typen zu unterscheiden:

Alle C++-Typen
- ***Definierte Typen***
  - Vordefinierte Typen (built in types, predefined types)
    - Arithmetische Typen (arithmetic types)
    - Der leere Typ
  - Vom Programmierer definierte Typen (user defined types)
    - Klassentypen (class types)
    - Vereinigungstypen (union types)
    - Aufzählungstypen (enumeration types)
- ***Konstruierte Typen***
  - Reihungstypen (array types)
  - Unterprogrammtypen (function types)
  - Adresstypen (pointer types)
  - Referenztypen (reference types)
  - Elementadresstypen (types of pointers to non-static class members)

Leider wird diese Unterscheidung vom C++-Standard nicht unterstützt: Es fehlen offizielle Begriffe für die Typarten, die hier als ***definierte Typen*** und als ***konstruierte Typen*** bezeichnet werden.

Auch in Java gibt es ***konstruierte Typen***: Zu jedem Typ `T` gibt es u.a. die Reihungstypen `T[]`, `T[][]`, `T[][][]`, ... etc. Diese Reihungstypen "bekommt man vom Compiler geschenkt" und ***man kann sie nicht vereinbaren***. Weil die Namen dieser Typen aus einem Typnamen und gewissen Sonderzeichen (`"["` und `"]"`) zusammengesetzt oder ***konstruiert*** werden, werden diese Typen hier als ***konstruierte Typen*** bezeichnet.

In Java gibt es nur ***eine Art*** von konstruierten Typen, nämlich die ***Reihungstypen***. In C++ gibt es dagegen ***fünf verschiedene Arten*** (siehe oben die letzte Übersicht über die C++-Typen). Die folgenden Beispiele sollen einen ersten Eindruck davon vermitteln, um was für Typen es sich dabei handelt:

Name des Typs	***Erläuterung***
int[]	Eindimensionale Reihung von int-Variablen
string[][]	Zweidimensionale Reihung von string-Variablen
const float[][][]	Dreidimensionale Reihung von float-Konstanten
int *	Adresse von int-Variable
const int *	Adresse von int-Konstante
int * const	Konstanter Adresse von int-Variable
const int * const	Konstanter Adresse von int-Konstante
int * []	Eindimensionale Reihung von Adressen auf int-Variablen
int (*) []	Adresse von eindimensionale Reihung von int-Variablen
bool (*) (int, int)	Adresse von Funktion mit bool-Ergebnis und zwei int-Parametern
void (*) ()	Adresse von parameterlose Prozedur
int &	Referenz auf int-Variable

Name des Typs	*Erläuterung*
const int &	Referenz auf int-Konstante
int * &	Referenz auf Adresse von int-Variable

In Java ist genau festgelegt, welche Werte zu den einzelnen primitiven Typen (`byte`, `char`, `short`, `int`, ...) gehören. Der C++-Standard legt dagegen nur einige ***Typnamen*** fest, die jeder C++-Ausführer "kennen und akzeptieren" muss. Welche ***Werte*** zu den einzelnen Typen gehören, legt der C++-Standard ausdrücklich ***nicht*** fest. Manche C++-Ausführer stellen `int`-Werte mit 16 Bit dar, andere nehmen 32 Bit oder andere Bit-Zahlen. Bei manchen Ausführern sind die Typen `short` und `int` gleich, aber verschieden von `long`, bei anderen sind `int` und `long` gleich aber verschieden von `short` etc.

Das folgende Beispielprogramm gibt einige Typgrenzen des Gnu-Cygnus-Ausführers aus. Dabei werden die alten Kopfdateien `<climits>` und `<cfloat>` verwendet:

```
1   // Datei FundTypen02.cpp
2   /* ------------------------------------------------------------------------
3   Gibt die kleinsten und groessten Werte wichtiger fundamentaler Typen aus.
4   ------------------------------------------------------------------------ */
5   #include <iostream> // fuer cin und cout
6   #include <iomanip>  // fuer setw
7   #include <climits>  // fuer CHAR_MIN, CHAR_MAX, ..., ULONG_MAX
8   #include <cfloat>   // fuer FLT_MAX,  DBL_MAX, LDBL_MAX
9   using namespace std;
10
11  int main() {
12     const int L = 15;
13     cout << "FundTypen02: Jetzt geht es los!"     << endl;
14     cout << "-------------------------------"     << endl; // Typ:
15     cout << " CHAR_MIN: " << setw(L) <<  CHAR_MIN << endl; // char
16     cout << " CHAR_MAX: " << setw(L) <<  CHAR_MAX << endl; // char
17     cout << "SCHAR_MIN: " << setw(L) << SCHAR_MIN << endl; // signed   char
18     cout << "SCHAR_MAX: " << setw(L) << SCHAR_MAX << endl; // signed   char
19     cout << "UCHAR_MAX: " << setw(L) << UCHAR_MAX << endl; // unsigned char
20     cout << "-------------------------------"     << endl;
21     cout << " SHRT_MIN: " << setw(L) <<  SHRT_MIN << endl; // (signed) short
22     cout << " SHRT_MAX: " << setw(L) <<  SHRT_MAX << endl; // (signed) short
23     cout << "USHRT_MAX: " << setw(L) << USHRT_MAX << endl; // unsigned short
24     cout << "-------------------------------"     << endl;
25     cout << "  INT_MIN: " << setw(L) <<   INT_MIN << endl; // (signed) int
26     cout << "  INT_MAX: " << setw(L) <<   INT_MAX << endl; // (signed) int
27     cout << " UINT_MAX: " << setw(L) <<  UINT_MAX << endl; // unsigned int
28     cout << "-------------------------------"     << endl;
29     cout << " LONG_MIN: " << setw(L) <<  LONG_MIN << endl; // (signed) long
30     cout << " LONG_MAX: " << setw(L) <<  LONG_MAX << endl; // (signed) long
31     cout << "ULONG_MAX: " << setw(L) << ULONG_MAX << endl; // unsigned long
32     cout << "-------------------------------"     << endl;
33     cout << "  FLT_MIN: " << setw(L) <<   FLT_MIN << endl; // float
34     cout << "  FLT_MAX: " << setw(L) <<   FLT_MAX << endl; // float
35     cout << "  DBL_MIN: " << setw(L) <<   DBL_MIN << endl; // double
36     cout << "  DBL_MAX: " << setw(L) <<   DBL_MAX << endl; // double
37     cout << " LDBL_MIN: " << setw(L) <<  LDBL_MIN << endl; // long double
38     cout << " LDBL_MAX: " << setw(L) <<  LDBL_MAX << endl; // long double
39     cout << "-------------------------------"     << endl;
40  } // main
41  /* ------------------------------------------------------------------------
42  Ausgabe des Programms FundTypen02 (compiliert mit dem Gnu-Cygnus-C++-Com-
43  piler Version 3.2 unter Windows):
44
```

```
45  FundTypen02: Jetzt geht es los!
46  ------------------------------
47   CHAR_MIN:             -128
48   CHAR_MAX:              127
49  SCHAR_MIN:             -128
50  SCHAR_MAX:              127
51  UCHAR_MAX:              255
52  ------------------------------
53   SHRT_MIN:           -32768
54   SHRT_MAX:            32767
55  USHRT_MAX:            65535
56  ------------------------------
57    INT_MIN:      -2147483648
58    INT_MAX:       2147483647
59   UINT_MAX:       4294967295
60  ------------------------------
61   LONG_MIN:      -2147483648
62   LONG_MAX:       2147483647
63  ULONG_MAX:       4294967295
64  ------------------------------
65    FLT_MIN:      1.17549e-38
66    FLT_MAX:      3.40282e+38
67    DBL_MIN:     2.22507e-308
68    DBL_MAX:     1.79769e+308
69   LDBL_MIN:     3.3621e-4932
70   LDBL_MAX:   1.18973e+4932
71  ------------------------------
```

Das folgende Beispielprogramm gibt einige Typgrenzen des **Borland**-Ausführers aus. Dabei wird die moderne Kopfdatei <**limits**> verwendet:

```
1   // Datei FundTypen03.cpp
2   /* ------------------------------------------------------------------------
3   Mit Hilfe der Schablone numeric_limits werden fuer verschiedene arithme-
4   tische Typen (int, short, long, ...) Minimum, Maximum und Anzahl der
5   Binaerziffern ausgegeben.
6   ------------------------------------------------------------------------ */
7   #include <iostream> // fuer cout
8   #include <iomanip>  // fuer setw
9   #include <limits>   // fuer numeric_limits
10  using namespace std;
11  // ------------------------------------------------------------------------
12  int const L = 15;
13  // Eine Unterprogrammschablone gibGrenzenAus mit einem Typ-Parameter:
14  template<typename TYP>
15  void gibGrenzenAus(char * text) {
16     cout << text;
17     cout << setw(L) << numeric_limits<TYP>::min()  << ", ";
18     cout << setw(L) << numeric_limits<TYP>::max()  << ", ";
19     cout << setw(L) << numeric_limits<TYP>::digits << endl;
20  } // template gibGrenzenAus
21
22  // Spezialisierung der Schablone gibGrenzenAus fuer den Typ char:
23  template<>
24  void gibGrenzenAus<char>(char * text) {
25     cout << text;
26     cout << setw(L) << (int) numeric_limits<char>::min()  << ", ";
27     cout << setw(L) << (int) numeric_limits<char>::max()  << ", ";
28     cout << setw(L) << (int) numeric_limits<char>::digits << endl;
29  }
30
31  // Spezialisierung der Schablone gibGrenzenAus fuer den Typ unsigned char:
```

```
32  template<>
33  void gibGrenzenAus<unsigned char>(char * text) {
34     cout << text;
35     cout << setw(L) << (int) numeric_limits<unsigned char>::min()  << ", ";
36     cout << setw(L) << (int) numeric_limits<unsigned char>::max()  << ", ";
37     cout << setw(L) << (int) numeric_limits<unsigned char>::digits << endl;
38  }
39  // ------------------------------------------------------------------------
40  int main() {
41     // Ueberschriftenzeile ausgeben:
42     cout << "Typ              " << setw(L) << "Minimum" <<
43        setw(17) << "Maximum" << setw(17) << "Binaerziffern" << endl << endl;
44
45     // Von verschiedenen arithmetischen Typen Min, Max etc. ausgeben:
46     gibGrenzenAus<char>          ("char:           ");
47     gibGrenzenAus<wchar_t>       ("wchar_t:        ");
48     gibGrenzenAus<short>         ("short:          ");
49     gibGrenzenAus<int>           ("int:            ");
50     gibGrenzenAus<long>          ("long:           ");
51     gibGrenzenAus<unsigned char> ("unsigned char:  ");
52     gibGrenzenAus<unsigned short>("unsigned short: ");
53     gibGrenzenAus<unsigned int>  ("unsigned int:   ");
54     gibGrenzenAus<unsigned long> ("unsigned long:  ");
55     gibGrenzenAus<float>         ("float:          ");
56     gibGrenzenAus<double>        ("double:         ");
57     gibGrenzenAus<long double>   ("long double:    ");
58     gibGrenzenAus<bool>          ("bool:           ");
59  } // main
60  /* ------------------------------------------------------------------------
61  Ausgabe des Programms FundTypen03 (compiliert mit dem Borland
62  C++-Compiler Version 5.5 unter Windows98):
63
64  Typ                     Minimum          Maximum    Binaerziffern
65
66  char:                      -128,             127,               7
67  wchar_t:                      0,           65535,              15
68  short:                   -32768,           32767,              15
69  int:                -2147483648,      2147483647,              31
70  long:               -2147483648,      2147483647,              31
71  unsigned char:                0,             255,               8
72  unsigned short:               0,           65535,              16
73  unsigned int:                 0,      4294967295,              32
74  unsigned long:                0,      4294967295,              32
75  float:              1.17549e-38,     3.40282e+38,              24
76  double:            2.22507e-308,    1.79769e+308,              53
77  long double:       3.3621e-4932,   1.18973e+4932,              64
78  bool:                         0,               1,               1
79  --------------------------------------------------------------------- */
```

Typen sind ***Baupläne für Variablen***. Um auch komplizierte Typen zu verstehen und sich die nach ihnen gebauten Variablen anschaulich vorstellen zu können ist es günstig, mit der ***Bojendarstellung*** für Variablen vertraut zu sein. Sie wurde im Zusammenhang mit der Sprache Algol68 entwickelt und wird im nächsten Abschnitt beschrieben.

6 Die Bojendarstellung für Variablen

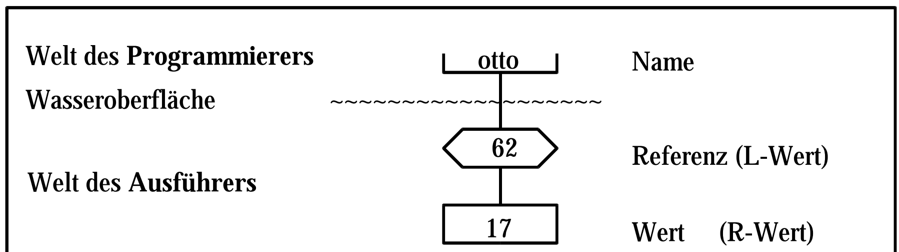
Das Konzept einer ***Variablen***, deren Wert man beliebig oft (durch Zuweisungen und andere Anweisungen) ***verändern*** kann, ist das wichtigste Grundkonzept der meisten Programmiersprachen. Dieses Konzept soll hier genauer beschrieben werden.

Def.: Eine ***Variable*** besteht aus mindestens ***zwei*** Teilen, einer ***Adresse*** und einem ***Wert***. Einige Variablen haben zusätzlich noch einen (oder mehrere) ***Namen*** und/oder einen ***Zielwert***.

Als ***Bojen*** werden hier bestimmte ***graphische Darstellungen von Variablen*** bezeichnet. Bojen eignen sich besonders gut dazu, sich den Unterschied zwischen verschiedenen Arten von Variablen anschaulich klar zu machen. Als erstes Beispiel hier die Definition einer `int`-Variablen:

```
1   int otto = 17;
```

Als Boje dargestellt sieht die Variable `otto` etwa so aus:



Diese Boje stellt eine Variable mit dem Namen `otto`, der Adresse `<62>` und dem Wert `17` dar. Der ***Name*** ist der einzige Teil, den der Programmierer direkt zu sehen bekommt. Deshalb gehört der ***Name*** zur ***Welt des Programmierers*** über der "Wasseroberfläche". Die ***Adresse*** wird vom Ausführer festgelegt und nur er kann den ***Wert*** der Variablen direkt manipulieren. Deshalb gehören die ***Adresse*** und der ***Wert*** der Variablen zur ***Welt des Ausführers*** unter der "Wasseroberfläche".

Man beachte, dass ***Adresseen*** immer in ***6***-eckigen Kästchen, alle anderen ***Werte*** dagegen in ***4***-eckigen Kästchen gezeichnet werden. Im Text notieren wir hier ***Adresseen*** entsprechend in spitzen Klammern (z. B. `<62>`), um sie deutlich von ***Ganzzahlwerten*** (z. B. `17`) zu unterscheiden.

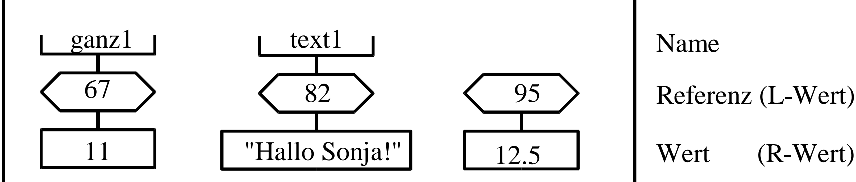
In den folgenden Beispielen werden jeweils ein paar Variablen ***vereinbart*** und dann ***als Bojen dargestellt***. Einige der dabei verwendeten Typen werden später genauer behandelt.

6.1 Bojen für Variablen definierter Typen

Die Bojen aller Variablen, die zu einem ***definierten*** Typ gehören, bestehen aus einem ***6-Eck*** für die ***Adresse*** und einem ***4-Eck*** für den ***Wert*** (und eventuell aus einem "***Paddelboot***" für den ***Namen***):

```
1   int    ganz1 = 11;
2   string text1 = "Hallo Sonja!";
3   new double(12.5);  // so erlaubt, aber nicht empfehlenswert!
```

Die Variablen `ganz1` und `text1` und die namenlose `double`-Variable sehen als Bojen dargestellt etwa so aus:



6.2 Bojen für zwei Adressvariablen

Adresstypen gehören in C++ zu den ***konstruierten*** Typen. Ihre Namen enden mit einem ***Sternchen*** (`int *`, `string *`, `int **` etc.). Das besondere an Adressvariablen: Ihr ***Wert*** ist kein "gewöhnlicher Wert in einem 4-eckigen Kästchen", sondern eine ***Adresse*** (in einem 6-eckigen Kästchen). Diese Adresse ist entweder gleich `NanA` ("not an address", alias `NULL`, dann zeigt die Adressvariable auf keinen Zielwert) oder verschieden von `NanA` (dann zeigt die Adressvariable auf einen Zielwert).

```
4   int    * zai1 = new int(12);                // Adresse von    int-Variable
5   string * zas1 = new string("Wie geht es?"); // Adresse von string-Variable
6   string * zas2 = NanA;                       // alias NULL, alias 0
```

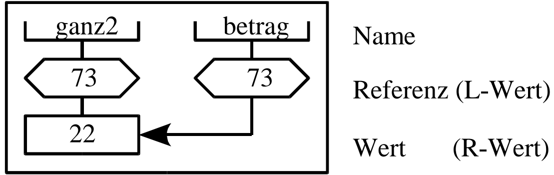
zai1 zas1 zas2 Name
55 12 32 Referenz (L-Wert)
83 77 NanA Wert (R-Wert)
12 "Wie geht's?" Zielwert

Für ***Java*** gilt: Die Bojen von ***Variablen eines Java-Referenztyps*** (z. B. die Bojen von `String`-Variablen) sehen so aus wie die Bojen von ***Adressvariablen in C++***. Einer `String`-Variablen in Java entspricht also nicht etwa eine `string`-Variable in C++, sondern eine Variable des Typs `string*` (Adresse von `string`).

6.3 Bojen für zwei Variablen mit gleichen Adressen und identischem Wert

Referenztypen in C++ (nicht zu verwechseln mit Referenztypen in Java) gehören ebenfalls zu den ***konstruierten*** Typen. Ihre Namen enden mit einem ***Ampersand*** (`int &`, `string &` etc.). Im Gegensatz zu allen anderen Variablen ***müssen*** Referenzvariablen vom Programmierer ***initialisiert*** werden, z. B. mit einer anderen Variablen, und sie bekommen dann nicht den ***Wert*** dieser Variablen sondern ihre ***Adresse***:

```
7   int    ganz2  = 22;
8   int & betrag = ganz2;
```

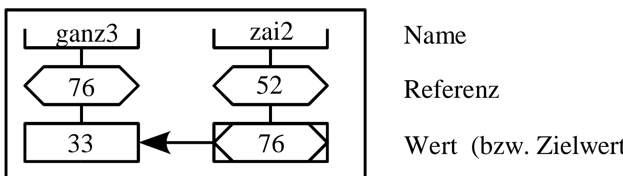


Hier wird die Variable `betrag` vom Typ ***Referenz auf*** `int` mit der `int`-Variablen `ganz2` initialisiert. Dadurch bekommt die Variable `betrag` die gleiche Adresse (`<73>`) wie `ganz2` und die beiden Variablen haben dadurch ***denselben Wert*** (***nicht***: zwei gleiche Werte!).

6.4 Bojen für zwei Variablen mit identischem Wert bzw. Zielwert

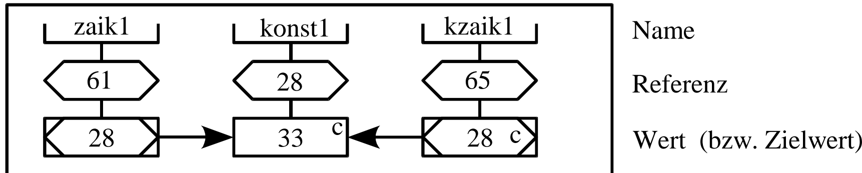
Ist `x` irgendeine Variable, dann bezeichnet `&x` die Adresse dieser Variablen. Eine solche Adresse kann man z. B. als ***Wert einer Adressvariablen*** verwenden:

```
9   int   ganz3 = 33;
10  int * zai2  = &ganz3; // Adresse von int
```



6.5 Bojen für Konstanten, Adressen von Konstanten und konstante Adressen

```
1   const int         konst1 = 33;      //                    int-Konstante
2   const int *       zaik1  = &konst1; //        Adresse von int-Konstante
3   const int * const kzaik1 = &konst1; // konst. Adresse von int-Konstante
```

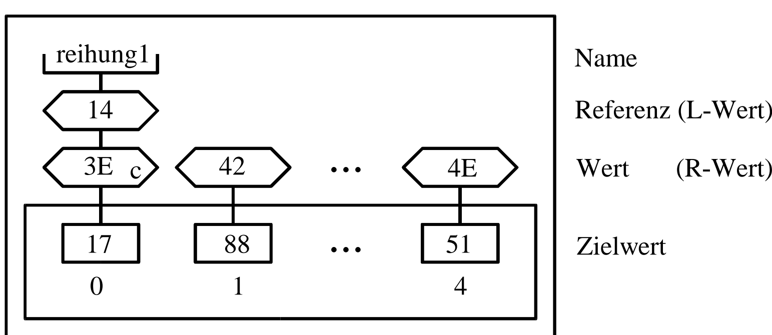


Der kleine Buchstabe `c` kennzeichnet Kästchen, denen der Programmierer normalerweise neue Werte zuweisen kann, die er hier aber wegen einer Angabe von `const` in der entsprechenden Variablenvereinbarung nicht verändern darf. Die ***Adresse*** einer Variablen ist grundsätzlich ***nicht*** veränderbar. Das ist so ***selbstverständlich***, dass man sie meist ***nicht*** mit einem `c` kennzeichnet.

6.6 Boje für eine Reihung

In C++ ist eine ***Reihungsvariable*** (array variable) kaum mehr als eine ***konstante Adresse*** der ersten Komponente der Reihung. Die Komponenten der Reihung sind (namenlose) Variablen, deren Adressen sich jeweils um die Länge einer Komponenten (gemessen in Bytes) unterscheiden. Bei vielen C++-Ausführern belegt eine `int`-Variable 4 Bytes.

```
1   int reihung1[5] = {17, 88, 35, 67, 51};
```



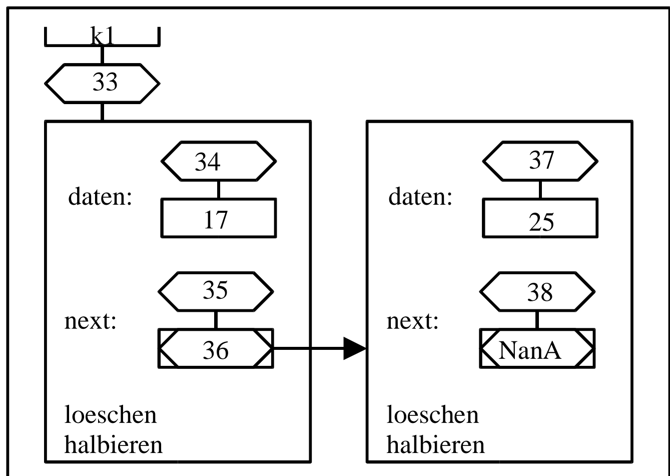
6.7 Bojen für Objekte

Die Objekte einer Klasse sollte man je nach Zusammenhang "ausführlich und vollständig" oder "ein bisschen vereinfacht" darstellen. Betrachten wir als Beispiel die folgenden Vereinbarungen:

```
1   struct Knoten {
2      // Klassenelemente:
3      Knoten(int daten=0, Knoten * next=NanA) : daten(daten), next(next) {}
4
5      // Objektelemente:
6      int      daten;
7      Knoten * next;
8      int  loeschen() {int erg=daten; daten=0; return erg;}
9      void halbiere() {daten /= 2;}
10  }; // struct Knoten
11
12  Knoten k1(17, new Knoten(25, NULL));
```

In C++ kann man die Vereinbarung einer Klasse wahlweise mit dem Schlüsselwort `struct` oder `class` beginnen. Verwendet man `struct`, so sind alle nicht anders gekennzeichneten Element öffentlich (`public`), wenn man `class` verwendet dagegen privat (`private`).

Die Klasse `Knoten` enthält 1 Klassenelement (einen ***Konstruktor***), 2 Objektattribute (`daten` und `next`) und 2 Objektmethode n (`loeschen` und `halbieren`). Hier eine "ausführliche und vollständige" Darstellung der Variablen `k1` als Boje:



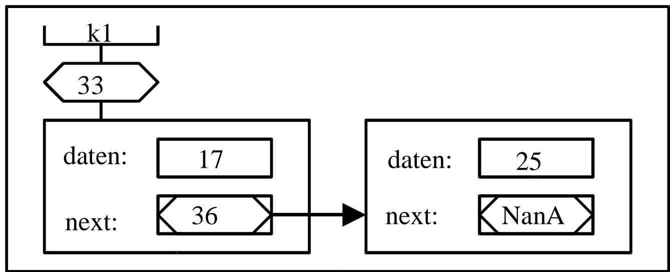
Man sieht hier 2 `Knoten`-Variablen. Die erste hat einen ***Namen*** (`k1`) und die ***Adresse*** `<33>`, die zweite hat keinen einfachen Namen, aber die ***Adresse*** `<36>`. Diese zweite `Knoten`-Variable kann man mit dem zusammengesetzten Namen `*(k1.next)` ("das, worauf `k1.next` zeigt") bezeichnen.

Die ***Attribute*** einer Objekt-Variablen sind selbst auch ***Variablen*** und bestehen (wie alle Variablen) aus einer ***Adresse*** und einem ***Wert***. Diese Attribut-Variablen haben aber ***keine einfachen Namen***. Man kann sie über die zusammengesetzten Namen `k1.daten`, `k1.next`, `k1.next->daten` und `k1.next->next` bezeichnen. Die Variable `k1.daten` hat im Beispiel die ***Adresse*** `<34>` und den ***Wert*** `17`, die Variable `k1.next->next` hat die ***Adresse*** `<38>` und den Wert `NanA` ("not an address", alias `NULL`, alias 0) etc.

Dem Namen `daten` ist ***keine*** (eindeutige) ***Adresse*** zugeordnet. Daraus folgt, dass ein Ausdruck wie `&daten` ("Adresse von Daten") verboten und `daten` somit ***kein Variablen-Name*** ist. Deshalb darf `daten` auch nicht in einem "Paddelboot" stehen wie z. B. der Variablen-Name `k1` (dem eindeutig die

Adresse `<33>` zugeordnet ist). Entsprechendes gilt auch für den Namen `next`. Man bezeichnet `daten` und `next` als ***Attribut-Namen*** oder als ***relative Namen*** (weil sie nur ***relativ*** zu einer bestimmten Variablen eine Bedeutung haben), um sie von ***Variablen-Namen*** zu unterscheiden.

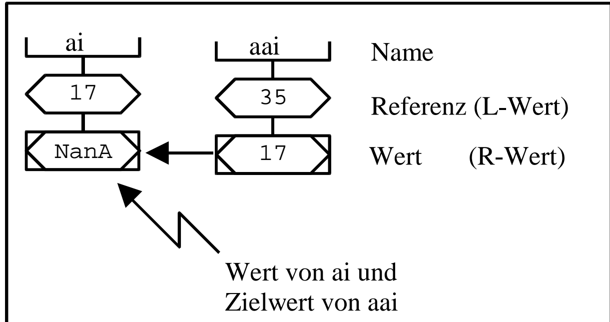
In obiger Bojen-Darstellung der beiden Knoten-Variablen sind auch die ***Objektmethoden*** (`loeschen` und `halbieren`) eingezeichnet. Die kann man meistens ***weglassen***. Die ***Adresseen*** der ***Attribute*** (`<34>`, `<35>`, `<37>`, `<38>`) kann man ebenfalls ***weglassen***, wenn man nicht gerade Ausdrücke wie `&k1.daten`, `&k1.next`, `&k1.next->daten` oder `&k1.next->next` erklären will. Hier eine vereinfachte Bojen-Darstellung derselben Variablen wie oben:



6.8 Eine Adressvariable, die auf NanA (alias NULL) zeigt

Im folgenden Beispiel zeigt die Adressvariable `aai` auf `NanA`:

```
1   int *    ai = NanA; // Eine Adressvariable mit dem Wert NanA
2   int * * aai = &zi;  // Eine Adressvariable, die auf NanA zeigt
```



Man beachte den feinen Unterschied: Die Variable `ai` ***hat den Wert*** `NanA` und zeigt somit auf ***keinen*** Zielwert. Die Variable `aai` (vom Typ ***Adresse von Adresse von*** `int`) hat den ***Wert*** `[<17>]` und den ***Zielwert*** `NanA`, d. h. `aai` "zeigt auf `NanA`" (oder: auf einen `NanA`-Wert).

Es kommt sehr häufig vor, dass eine Adressvariable ***den Wert NanA hat*** (wie oben `ai`). Dass eine Adressvariable ***auf NanA zeigt*** (wie oben `aai`) ist dagegen ***sehr selten***. Indem man die beiden Aussagen ***hat den Wert NanA*** und ***zeigt auf NanA*** möglichst häufig verwechselt und vermischt, kann man seine Hörer/Leser (und möglicherweise sogar sich selbst) sehr wirkungsvoll verwirren.

Mit Hilfe von Bojen kann man auch die Unterschiede und Entsprechungen zwischen Java-Typen und C++-Typen veranschaulichen.

C++-Typen und die zugehörigen ***Bojen***:

Definierte Typen in C++	**Adresstypen in C++**
z. B. **int**	z. B. **int ***
z. B. **string**	z. B. **string ***

Java-Typen und die zugehörigen ***Bojen***:

Primitive Typen in Java	**Referenztypen in Java**
z. B. **int**	**---**
---	z. B. **String**

7 Konstruierte Typen

Der fundamentale Typ `int`, der Klassentyp `string` und jeder ***vom Programmierer*** definierte (`enum`- oder Klassen-) Typ sind Beispiele für ***definierte Typen***. Zu jedem ***definierten*** Typ gibt es eine ganze Reihe ***konstruierter*** Typen, deren Namen aus dem Namen des definierten Typs, dem Schlüsselwort `const` und bestimmten Sonderzeichen (`*`, `&`, `[ , ]`) "konstruiert werden". Das folgende Beispielprogramm demonstriert alle Typen, deren Namen aus dem Typnamen `int`, dem Schlüsselwort `const` und ***einem*** Sternchen (`*`) konstruiert werden können.

```
1   // Datei KonstruierteTypen01.cpp
2   /* ------------------------------------------------------------------------
3   Einige konstruierte Typen mit "int" im Namen:          |Alternativer Name:
4                                                          |
5   const int          : int-Konstante                     | int const
6         int *        : Adresse von int-Variable          |
7   const int *        : Adresse von int-Konstante         | int const *
8         int * const  : Konstanter Adresse von int-Variable |
9   const int * const  : Konstanter Adresse von int-Konstante| int const * const
10  --------------------------------------------------------------------- */
11  #include <iostream>
12  using namespace std;
13
```

```
14  int main() {
15     cout << "KonstruierteTypen01: Jetzt geht es los!" << endl;
16     // ------------------------------------------------------------------
17     // Ein paar Variablen und Konstanten vereinbaren:
18           int         i1    = 11;   // int-Variable
19           int         i2    = 11;   // int-Variable
20
21     const int         ki1   = 12;   // int-Konstante
22     const int         ki2   = 12;   // int-Konstante
23
24           int *       zi1   = &i1;  // Adresse von int-Variable
25           int *       zi2   = &i2;  // Adresse von int-Variable
26  //       int *       zi3   = &ki1; // Typfehler bei Initialisierung
27
28     const int *       zik1  = &i1;  // Adresse von int-Konstante
29     const int *       zik2  = &ki1; // Adresse von int-Konstante
30
31           int * const kzi1  = &i1;  // Konstante Adresse von int-Variable
32           int * const kzi2  = &i2;  // Konstante Adresse von int-Variable
33  //       int * const kzi3  = &ki1; // Typfehler bei Initialisierung
34
35     const int * const kzik1 = &i1;  // Konstante Adresse von int-Konstante
36     const int * const kzik2 = &ki1; // Konstante Adresse von int-Konstante
37     // ------------------------------------------------------------------
38     // Ein paar Variablen und Konstanten ausgeben:
39                  cout << " i1   : " <<  i1   << endl;
40
41                  cout << "*zi1  : " << *zi1  << endl;
42                  cout << " zi1  : " <<  zi1  << endl;
43
44                  cout << "*zik1 : " << *zik1 << endl;
45                  cout << " zik1 : " <<  zik1 << endl;
46
47                  cout << "*kzi1 : " << *kzi1 << endl;
48                  cout << " kzi1 : " <<  kzi1 << endl << endl;
49     // ------------------------------------------------------------------
50     // Ein paar Variablen veraendern und erneut ausgeben:
51      i1    = 21;   cout << " i1   : " <<  i1   << endl;
52  //  ki1   = 22;
53
54     *zi1   = 23;   cout << "*zi1  : " << *zi1  << endl;
55      zi1   = zi2;  cout << " zi1  : " <<  zi1  << endl;
56
57  // *zik1  = 24;
58                  cout << "*zik1 : " << *zik1 << endl;
59      zik1  = zik2; cout << " zik1 : " <<  zik1 << endl;
60
61     *kzi1  = 25;   cout << "*kzi1 : " << *kzi1 << endl;
62                  cout << " kzi1 : " <<  kzi1 << endl;
63  //  kzi1  = kzi2;
64
65  // *kzik1 = 26;
66  //  kzik1 = kzik2;
67     // ------------------------------------------------------------------
68     cout << "KonstruierteTypen01: Das war's erstmal!" << endl;
69  } // main
70  /* ---------------------------------------------------------------------
71  Fehlermeldungen des Compilers, wenn die entsprechenden Zeilen keine
72  Kommentare sind:
73
74  KonstruierteTypen01.cpp:25: initialization to `int *' from `const int *'
75                              discards qualifiers
```

```
76  KonstruierteTypen01.cpp:32: initialization to `int *const' from `const int *'
77                              discards qualifiers
78  KonstruierteTypen01.cpp:48: assignment of read-only variable `ki1'
79  KonstruierteTypen01.cpp:53: assignment of read-only location
80  KonstruierteTypen01.cpp:57: assignment of read-only variable `kzi1'
81  KonstruierteTypen01.cpp:59: assignment of read-only location
82  KonstruierteTypen01.cpp:60: assignment of read-only variable `kzik1'
83  ------------------------------------------------------------------------------
84  Ausgabe des Programms KonstruierteTypen01:
85
86  KonstruierteTypen01: Jetzt geht es los!
87   i1   : 11
88  *zi1  : 11
89   zi1  : 0x258fc04
90  *zik1 : 11
91   zik1 : 0x258fc04
92  *kzi1 : 11
93   kzi1 : 0x258fc04
94
95   i1   : 21
96  *zi1  : 23
97   zi1  : 0x258fc00
98  *zik1 : 23
99   zik1 : 0x258fbfc
100 *kzi1 : 25
101  kzi1 : 0x258fc04
102 KonstruierteTypen01: Das war's erstmal!
103 --------------------------------------------------------------------------- */
```

Aufgabe: Stellen Sie möglichst viele der in diesem Programm vereinbarten Variablen und Konstanten als ***Bojen*** dar.

Das folgende Beispielprogramm demonstriert weitere ***konstruierte*** Typen:

```
1   // Datei KonstruierteTypen02.cpp
2   /* ------------------------------------------------------------------------
3   Einige konstruierte Typen mit "float" im Namen:
4   float *        Adresse von                        float-Variable
5   float []       Eindimensionale  Reihung von       float-Variablen
6   float [][]     Zweidimensionale Reihung von       float-Variablen
7   float * []     Eindim. Reihung von Adressen  von float-Variablen
8   float (*) []   Adresse von eindim. Reihungen von float-Variablen
9   --------------------------------------------------------------------- */
10  #include <iostream>
11  using namespace std;
12
13  int main() {
14     cout << "Programm KonstruierteTypen02: Jetzt geht es los!" << endl;
15     //---------------------------------------------------------------------
16     // Die Typen float und float* (Adresse von float-Variable)
17     float  otto  = 44.44;
18     float *anna;                      // Die Adresse anna zeigt "irgendwohin"
19     float *bert  = new float(11.11);
20     float *celia = new float(22.22), dora = 33.33, *emil = &otto;
21
22     cout << "abcde:   "
23          << anna << " " << *bert << " " << *celia << " "
24          << dora << " " << *emil << " " <<  endl;
25     //---------------------------------------------------------------------
26     // Der Typ float [] (Eindimensionale Reihungen von float-Variablen)
27     float fritz[3] = {11.11, 22.22}; // fritz[2] wird 0.0
28     cout << "fritz:   ";
```

```
29     for (int i=0; i < 3; i++) {
30        cout << fritz[i] << " ";
31     }
32     cout << endl;
33     //------------------------------------------------------------------------
34     // Der Typ float[][] (Zweidimensionale Reihungen von float-Variablen)
35     float gerd[2][3] = {{11.11, 21.21, 31.31}, {12.12, 22.22, 32.32}};
36     for (int i1=0; i1 < 2; i1++) {
37        cout << "gerd[" << i1 << "]: ";
38        for (int i2=0; i2 < 3; i2++) {
39           cout << gerd[i1][i2] << " ";
40        }
41        cout << endl;
42     }
43     //------------------------------------------------------------------------
44     // Der Typ float * [] (Eindim. Reihungen von Adressen von float-Variable)
45     float * helga[3] = {bert, &otto, new float(55.55)};
46     cout << "helga:   ";
47     for (int i=0; i < 3; i++) {
48        cout << *helga[i] << " ";
49     }
50     cout << endl;
51     //------------------------------------------------------------------------
52     // Der Typ float (*) [] (Adresse von eindim. Reihungen von float-Var.)
53     cout << "iris:    ";
54     float (*iris)[3] = &fritz;
55     for (int i=0; i < 3; i++) {
56        cout << (*iris)[i] << " ";
57     }
58     cout << endl;
59     //------------------------------------------------------------------------
60     cout << "Programm KonstruierteTypen02: Das war's erstmal!" << endl;
61  } // main
62  /* ------------------------------------------------------------------------
63  Ausgabe des Programms KonstruierteTypen02:
64
65  Programm KonstruierteTypen02: Jetzt geht es los!
66  abcde:   0x61005e7c 11.11 22.22 33.33 44.44
67  fritz:   11.11 22.22 0
68  gerd[0]: 11.11 21.21 31.31
69  gerd[1]: 12.12 22.22 32.32
70  helga:   11.11 44.44 55.55
71  iris:    11.11 22.22 0
72  Programm KonstruierteTypen02: Das war's erstmal!
73  ------------------------------------------------------------------------ */
```

Aufgabe: Stellen Sie möglichst viele der in diesem Programm vereinbarten Variablen und Konstanten als Bojen dar.

7.1 Reihungstypen (array types)

Reihungstypen gehören in C/C++ zu den ***konstruierten*** Typen (die man weder definieren ***muss*** noch ***kann***). Zugriffe auf Reihungen sind in C/***C++ schneller*** und ***fehlerträchtiger*** als in ***Java***. Wenn möglich, sollte man in C++-Programmen anstelle von "***alten C-Reihungen***" lieber "moderne" ***Objekte einer vector-Klasse*** ("C++-Reihungen") verwenden (siehe Abschnitt 9.1).

Beispiel für die Vereinbarung und Initialisierung einer Reihung:

```
1   int       ri1[3] = {35, 17, 22};
2   int const LIB    = sizeof(ri1);                 // Laenge in Bytes
3   int const LIK    = sizeof(ri1)/sizeof(ri1[0]); // Laenge in Komponenten
```

Eine C-***Reihung*** besteht nur aus ihren Komponenten und enthält ***keine*** zusätzlichen Attribute (wie etwa `length`) oder Methoden (wie etwa `size`). Dem ***Namen*** der Reihung ist die Anfangsadresse der ersten Komponente und die ***Länge der Reihung*** (gemessen in Bytes) "zugeordnet" und man kann diese Länge mit der Operation `sizeof` abfragen (natürlich nur an solchen Stellen eines Programms, an denen der Name sichtbar ist). Wenn eine `int`-Variable 4 Bytes belegt (wie bei vielen C++-Ausführern), dann hat im obigen Beispiel die Konstante `LIB` ("Länge in Bytes") den Wert `12` und die Konstante `LIK` ("Länge in Komponenten") den Wert `3`.

Die wichtigsten ***Eigenschaften*** von Reihungen (gilt nicht nur für C/C++, sondern für alle Programmiersprachen):

Eigenschaft R1: Der Zeitbedarf für einen Zugriff auf die `i`-te Komponente `r[i]` einer Reihung `r` ist unabhängig vom Index `i`.

Es dauert also (normalerweise) gleich lang, ob man auf die erste, auf die letzte oder auf irgendeine andere Komponente einer Reihung zugreift.

Eigenschaft R2: Der Zeitbedarf für den Zugriff auf eine Reihungskomponente `r[i]` ist unabhängig von der ***Länge*** der Reihung `r`.

Eigenschaft R3: Ein Zugriff auf eine Reihungskomponente `r[i]` kostet relativ ***wenig*** Zeit ("Reihungen sind sehr ***schnell***").

Eigenschaft R4: Beim Erzeugen einer Reihung wird ihre Länge festgelegt und kann danach nicht mehr verändert werden ("Reihungen sind aus Beton, nicht aus Gummi").

Speziell in C- und C++-Programmen sind Reihungen aber auch mit zwei Problemen verbunden:

Reihungsproblem 1: Beim Zugreifen auf die ***Komponenten*** einer Reihung findet ***keine Indexprüfung*** statt. Man kann also z. B. auf die 4. Komponente einer Reihung der Länge 3 zugreifen (oder auf die 400. Komponente einer solchen Reihung). Was dann genau passiert, hängt sehr von den näheren Umständen ab. Wenn man Glück hat, stürzt das Programm sofort ab und man bemerkt den Fehler. Wenn man Pech hat, passiert erstmal nichts auffälliges und der Fehler bleibt lange Zeit unentdeckt.

Reihungsproblem 2: Reihungen kann man ***nicht*** als ***Parameter*** an ***Unterprogramme*** übergeben. Wenn man es doch versucht, wird die Reihung in eine ***Adresse ihrer ersten Komponente*** umgewandelt und nur diese ***Adresse*** wird an das Unterprogramm übergeben. Mit der Adresse sind ***keine*** Informationen über die ***Länge*** der Reihung verbunden. Das zwingt den Programmierer meistens dazu, die Länge der Reihung als ***zusätzlichen Parameter*** an das Unterprogramm zu übergeben. Der Ausführer stellt ***keinen*** Fehler fest, wenn der Programmierer eine ***falsche*** Länge übergibt.

Diese Schwächen von C/C++-Reihungen (zusammen mit einem wenig sorgfältigen und/oder zu sehr auf Effizienz ausgerichteten Programmierstil) haben in Betriebssystemen, Browsern und anderen Programmen zu vielen Sicherheitslücken geführt. Ein Angreifer kann typische Lücken ausnützen, indem

er einen sogenannten ***Pufferüberlauf*** (buffer overflow) provoziert und damit bewirkt, dass bestimmte Befehle des Programms, die "hinter einer Reihung stehen", durch neue Befehle (des Angreifers) überschrieben werden. Es wäre sicher schwierig (aber auch interessant), abzuschätzen, wieviel Geld bisher verloren ging, weil in C/C++-Programmen bei Zugriffen auf Reihungen keine Indexprüfung stattfindet.

7.1.1 Eindimensionale Reihungen

Hier ein Beispielprogramm, in dem ein paar eindimensionale Reihungen vereinbart und bearbeitet werden. `"size_t"` ist ein Alias-Name für einen vorzeichenlosen Ganzzahltyp (z. B. `unsigned int`), der sich besonders gut für die Indizes von Reihungen eignet.

```
1   // Datei Reihungen10.cpp
2   /* ------------------------------------------------------------------------
3   Reihungen (arrays) und ihre Laengen.
4   ------------------------------------------------------------------------ */
5   #include <iostream>
6   #include <string>
7   #include <cstddef>  // fuer size_t
8   using namespace std;
9   // ------------------------------------------------------------------------
10  // Ein paar int-Reihungen der Laenge 3 vereinbaren:
11  int r01[3];                 // Explizite Laengenangabe, hier werden Kompo-
12                              // nenten automatisch mit 0 initialisiert
13  int r02[]  = {7, 2, 5};     // Implizite Laengenangabe durch Initialisierung
14  int r03[3] = {5, 7, 2};     // Explizite und implizite Laengenangabe
15
16  // Wie man es machen sollte:
17  size_t const LEN1 = 3;      // Laenge der folgenden Reihungen
18  int r04[LEN1];              // Explizite Laengenangabe
19  int r05[LEN1] = {2, 7, 5}; // Explizite und implizite Laengenangabe
20
21  // Wie es auch geht (die Laenge einer Reihung mit sizeof berechnen):
22  int r06[]     = {3, 9, 4, 7, 2};
23  size_t LEN2   = sizeof(r06)/sizeof(r06[0]);
24  // sizeof(X) liefert die Laenge von X gemessen in Bytes.
25
26  // Unvollstaendige Initialisierung (irrefuehrend, aber erlaubt):
27  int r07[5]    = {7, 3};     // Die Komponenten r07[2] bis r07[4] werden
28                              // automatisch mit 0 initialisiert.
29
30  // Nicht erlaubte Vereinbarungen von Reihungen:
31  /*
32  int r08[2]    = {1, 2, 3}; // Zuviele Werte,          verboten!
33  int r09[];                  // Keine Laenge angegeben, verboten!
34  int r10[0];                 // Leere Reihung,          verboten!
35  int r11[]     = {};         // Leere Reihung,          verboten!
36  */
37  // ------------------------------------------------------------------------
38  // Ein Upro zum Ausgeben von int-Reihungen der Laenge 3 (kaum nuetzlich):
39  void put3(string const text, int const * const r) {
40     // Gibt den text und 3 Komponenten der Reihung r in lesbarer Form aus:
41     cout << text + " ";
42     for (size_t i=0; i<3; i++) { // Typische Schleife zur Bearbeitung
43        cout << r[i] << " ";      // einer Reihung r
44     }
45     // sizeof auf einen Adress-Parameter anwenden bringt nicht viel:
46     cout << "sizeof(r): " << sizeof(r) << endl;
47  } // put3
```

```
48 // Um den Leser irrezufuehren kann man in der Parameterliste anstelle von
49 // "int const * const r" ("r ist eine konstante Adresse eines konstanten
50 // int-Wertes") auch "int const r[]" oder sogar "int const r[17]" angeben
51 // (dabei ist die Zahl 17 beliebig, der Ausfuehrer ignoriert sie).
52 // ------------------------------------------------------------------------
53 // Ein Unterprogramm zum Ausgeben von int-Reihungen beliebiger Laenge
54 void putN(string const text, int const * const r, const size_t laenge) {
55    // Gibt den text und laenge-viele Komponenten der Reihung r in lesbarer
56    // Form aus:
57    cout << text + " ";
58    for (size_t i=0; i<laenge; i++) {
59       cout << r[i]   << " "; // Syntax fuer Zugriff auf Komponenten von r
60 //    cout << i[r]   << " "; // Andere Syntax, gleiche Bedeutung
61 //    cout << *(r+i) << " "; // Andere Syntax, gleiche Bedeutung
62    }
63    cout << endl;
64 } // putN
65 // ------------------------------------------------------------------------
66 int main() {
67    cout << "Reihungen10: Jetzt geht es los!" << endl;
68
69    // In diesem *Unterprogramm* eine Reihung vereinbaren:
70    int r12[3];                 // Komponenten werden nicht initialisiert!
71
72    put3("r01:", r01);
73    put3("r02:", r02);
74    put3("r03:", r03);
75    put3("r04:", r04);
76    put3("r05:", r05);
77    put3("r06:", r06);          // Nur 3 der 5 Komponenten werden ausgegeben!
78    putN("r06:", r06, LEN2);    // Jetzt werden alle 5 Komponenten ausgegeben.
79    putN("r07:", r07, 5);       // Alle 5 Komponenten werden ausgegeben.
80    putN("r07:", r07, 7);       // 7 von 5 Komponenten (!) werden ausgegeben,
81                                // ganz uebler Fehler!
82    putN("r12:", r12, 3);       // Alle 3 Komponenten werden ausgegeben.
83
84    // Zuweisungen an eine Reihungsvariable:
85 // r01 = {1, 2, 3};            // Verboten!
86 // r01 = r02;                  // Verboten!
87
88    // Eine Initialisierung und eine Zuweisung die erlaubt sind:
89    int * p02 = r02; // Vereinbarung mit  Initialisierung, keine Zuweisung!
90    int * p03;       // Vereinbarung ohne Initialisierung.
91          p03 = r03; // Zuweisung,  keine Initialisierung!
92    put3("p02:", p02);
93    put3("p03:", p03);
94
95    cout << "Reihungen10: Das war's erstmal!" << endl;
96 } // main
97 /* ------------------------------------------------------------------------
98 Fehlermeldungen des Compilers (BCC 5.5), wenn die entsprechenden Zeilen
99 keine Kommentare sind:
100
101 Error E2225 Reihungen10.cpp 32: Too many initializers
102 Error E2449 Reihungen10.cpp 33: Size of 'r09' is unknown or zero
103 Error E2021 Reihungen10.cpp 34: Array must have at least one element
104 Error E2264 Reihungen10.cpp 35: Expression expected
105 Error E2188 Reihungen10.cpp 85: Expression syntax in function main()
106 Error E2277 Reihungen10.cpp 86: Lvalue required in function main()
107 ------------------------------------------------------------------------
108 Ausgabe des Programms Reihungen10:
109
```

```
110 Reihungen10: Jetzt geht es los!
111 r01: 0 0 0 sizeof(r): 4
112 r02: 7 2 5 sizeof(r): 4
113 r03: 5 7 2 sizeof(r): 4
114 r04: 0 0 0 sizeof(r): 4
115 r05: 2 7 5 sizeof(r): 4
116 r06: 3 9 4 sizeof(r): 4
117 r06: 3 9 4 7 2
118 r07: 7 3 0 0 0
119 r07: 7 3 0 0 0 2097184 1702521203
120 r12: 6684072 4224434 8
121 p02: 7 2 5 sizeof(r): 4
122 p03: 5 7 2 sizeof(r): 4
123 Reihungen10: Das war's erstmal!
124 ------------------------------------------------------------------------ */
```

Aufgabe: Stellen Sie einige der im Programm `Reihungen10` vereinbarten Variablen (`r01`, `r02`, ..., `p02`, `p03`) als ***Bojen*** dar.

7.1.2 Mehrdimensionale Reihungen

Hier ein Beispielprogramm, in dem ein paar ***mehrdimensionale*** (genauer: zweidimensionale) ***Reihungen*** vereinbart und bearbeitet werden.

```
1   // Datei Reihungen02.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert mehrdimensionale Reihungen und Unterprogramme zum Bearbeiten
4   solcher Reihungen.
5   Eine n-dimensionale Reihung hat n Laengen, in jeder Dimension eine. Eine
6   zweidimensionale Reihung mit den Laengen 2 und 5 enthaelt 10 Komponenten.
7   ------------------------------------------------------------------------ */
8   #include <iostream>
9   #include <cstddef>  // fuer den (vorzeichenlosen Ganzzahl-) Typ size_t
10  #include <string>
11  using namespace std;
12  // ------------------------------------------------------------------------
13  // Konstanten fuer die zwei Laengen einer zweidimensionalen Reihung:
14  size_t const DIM1 = 2; // 2 "Zeilen"
15  size_t const DIM2 = 4; // 4 "Spalten"
16  // ------------------------------------------------------------------------
17  void put1(string name, char r[][DIM2]) {
18     // Gibt die zweidimensionale char-Reihung r "in lesbarer Form" aus.
19     // DIM2: Die Laengen von r in allen Dimensionen ausser der ersten
20     // muessen hier angegeben und damit festgelegt werden.
21     cout << "put1(" << name << "):" << endl;
22     for (size_t i=0; i<DIM1; i++) { // Hier muss auch DIM1 bekannt sein!
23        cout << "  ";
24        for (size_t j=0; j<DIM2; j++) {
25           cout << r[i][j] << " ";
26        }
27        cout << endl;
28     } // for
29     cout << "--------------------" << endl;
30  } // put1
31  // ------------------------------------------------------------------------
```

```
32  void put5(string name, int r[][DIM2]) {
33     // Gibt die zweidimensionale char-Reihung r "in lesbarer Form" aus.
34     // DIM2: Die Laengen von r in allen Dimensionen ausser der ersten
35     // muessen hier angegeben und damit festgelegt werden.
36     cout << "put5(" << name << "):" << endl;
37     for (size_t i=0; i<DIM1; i++) { // Hier muss auch DIM1 bekannt sein!
38        cout << "  ";
39        for (size_t j=0; j<DIM2; j++) {
40           cout << r[i][j] << " ";
41        }
42        cout << endl;
43     } // for
44     cout << "---------------------" << endl;
45  } // put1
46  // ------------------------------------------------------------------------
47  void put2(string name, char * r, size_t laenge1, size_t laenge2) {
48     // r sollte auf eine zweidimensionale Reihung von char mit den Laengen
49     // laenge1 (erste Dimension) und laenge2 (zweite Dimension) zeigen.
50     // Diese zweidimensionale Reihung wird "in lesbarer Form" ausgegeben.
51     cout << "put2(" << name << "):" << endl;
52     for (size_t i=0; i<laenge1; i++) {
53        cout << "  ";
54        for (size_t j=0; j<laenge2; j++) {
55           cout << r[i*laenge2 + j] << " ";
56        }
57        cout << endl;
58     } // for
59     cout << "---------------------" << endl;
60  } // put2
61  // ------------------------------------------------------------------------
62  int main() {
63     // Vereinbarung einer zweidimensionalen Reihungen ohne Initialisierung.
64     // Beide Laengen muessen (explizit) angegeben werden:
65     char r1[3][5];
66     // "zweidimensionale Initialisierung" sieht gut aus:
67     char r2[DIM1][DIM2] = {{'7', '6', '5', '4'}, {'3', '2', '1', '0'}};
68     int  r5[2][4] = {{7, 6, 5, 4}, {3, 2, 1}};
69     // "eindimensionale Initialisierung" ist erlaubt:
70     char r3[DIM1][DIM2] = { '7', '6', '5', '4', '3', '2', '1', '0'};
71     // Nur die erste Laenge darf man implizit (per Initialisierung) angegen,
72     // alle weiteren Laengen muss man explizit angeben:
73     char r4[][DIM2] = {{'A', 'B', 'C', 'D'}, {'a', 'b', 'c', 'd'}};
74     // Falsche Vereinbarung, weil die 2. Laenge nicht explizit angegeben ist:
75  // char r6[][]      = {{'A', 'B', 'C', 'D'}, {'a', 'b', 'c', 'd'}};
76
77     // Zuweisungen zwischen Reihungs-Variablen sind nicht erlaubt (weil
78     // eine Reihungs-Variable eine Adress-Konstante ist):
79  // r2 = r3;
80
81     // Zugriff auf Komponenten einer zweidim. Reihung, normale Syntax:
82     r2[1][3] = 'X';
83     r4[1][3] = r2[1][3] + 1; // Was ist 'X' + 1 ?
84
85     // Reihungen ausgeben mit put1 put5 und put2:
86     cout << "A "; put1("r2", r2);
87     cout << "B "; put5("r5", r5);
88     cout << "C "; put2("r4", reinterpret_cast<char *>(r4), DIM1, DIM2);
89
90     // Zugriff auf Komponenten, exotische Syntax:
91     cout << "D r2[7]:          " << ((char *) r2)[7] << endl;
92     cout << "E r2[17, 1][3]: " << r2[17, 1][3]     << endl;
93  } // main
```

```
94  /* ------------------------------------------------------------------------
95  Fehlermeldungen des Gnu-Compilers, wenn die entsprechenden Zeilen keine
96  Kommentar sind:
97
98  Reihungen02.cpp: In function `int main()':
99  Reihungen02.cpp:75: error: declaration of `r6' as multidimensional array must
100    have bounds for all dimensions except the first
101 Reihungen02.cpp:75: error: assignment (not initialization) in declaration
102 Reihungen02.cpp:79: error: ISO C++ forbids assignment of arrays
103 ------------------------------------------------------------------------
104 Fehlermeldungen des Borland-Compilers, wenn die entsprechenden Zeilen kein
105 Kommentare sind:
106
107 Error E2453 D:\CPP\Reihungen02.cpp 75: Size of the type 'char[]' is unknown
108                                        or zero in function main()
109 Error E2277 D:\CPP\Reihungen02.cpp 79: Lvalue required in function main()
110 ------------------------------------------------------------------------
111 Ausgabe des Programms Reihungen02:
112
113 A put1(r2):
114   7 6 5 4
115   3 2 1 X
116 ---------------------
117 B put5(r5):
118   7 6 5 4
119   3 2 1 0
120 ---------------------
121 C put2(r4):
122   A B C D
123   a b c Y
124 ---------------------
125 D r2[7]:        X
126 E r2[17, 1][3]: X
127 --------------------------------------------------------------------- */
```

Merke: In Java gibt es ***mehrstufige*** Reihungen, ***mehrdimensionale*** Reihungen kann man aber als Objekte entsprechender Klassen "nachmachen" oder "vortäuschen". In C/C++ gibt es vor allem ***mehrdimensionale*** Reihungen, mit Hilfe von Adresstypen kann man aber auch ***mehrstufige*** Reihungen relativ leicht realisieren (siehe dazu auch die hier nicht wiedergegebenen Beispielprogramme `Reihungen04.cpp` und `Reihungen12.cpp`).

Mehrstufige Reihungen: Die Komponenten einer ***mehrstufigen*** Reihung `ms` des Typs `int[][]` sind Reihungen vom Typ `int[]`, auf die man mit Hilfe von ***einem*** Index zugreifen kann, etwa so: `ms[0]`, `ms[1]`, ... etc. Die ***elementaren Komponenten*** der Reihung `ms` sind vom Typ `int` und man muss ***zwei*** Indizes angeben, um auf sie zuzugreifen, etwa so: `ms[0][0]`, `ms[0][1]`, ..., `ms[3][2]`, ...

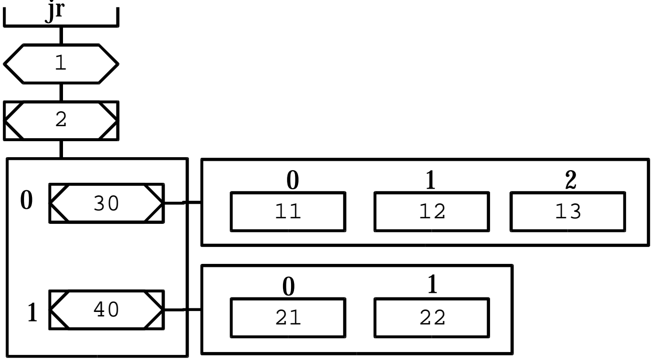
Mehrdimensionale Reihungen: Die Komponenten einer ***mehrdimensionalen*** Reihung `md` sind Variablen vom Typ `int` und man muss ***zwei*** Indizes angeben, um auf sie zuzugreifen, etwa so: `md[0][0]`, `md[0][1]`, ..., `md[3][2]`, ...

Mit Hilfe der Bojendarstellung kann man sich den Unterschied zwischen ***mehrdimensionalen*** und ***mehrstufigen*** Reihungen besonders anschaulich klar machen.

Eine in der Sprache Java vereinbarte ***zweistufige*** Reihung:

```
128  int[][] jr = { {11, 12, 13}, {21, 22} };
```

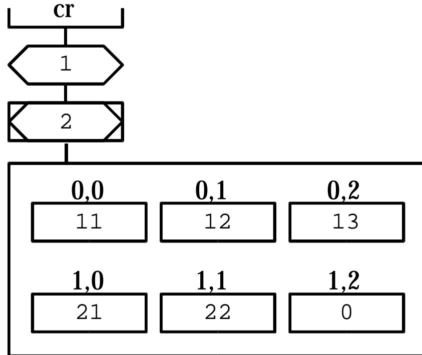
Als Boje dargestellt sieht die Reihung `jr` etwa so aus:



Eine in der Sprache C++ vereinbarte ***zweidimensionale*** Reihung:

```
129  int cr[2][3] =  { {11, 12, 13}, {21, 22} };
```

Als Boje dargestellt sieht die Reihung `cr` etwa so aus:



In der Vereinbarung der Reihung `cr` hat der Programmierung keinen Wert für die letzte Komponente `cr[1][2]` angegeben. Diese Komponente wird automatisch mit 0 initialisiert, wenn die Reihung direkt in einer ***Quelldatei*** vereinbart wird, bleibt aber uninitialisiert ("enthält irgendeinen Wert"), wenn die Reihung innerhalb eines ***Unterprogramms*** vereinbart wird.

Die Komponenten einer mehrdimensionalen Reihung stehen im Speicher eines heute üblichen Computers garantiert "***dicht hintereinander***". In C/C++ kann man desshalb z. B. eine ***zweidimensionale*** Reihung mit ***3 mal 5*** Komponenten auch als ***eindimensionale*** Reihung der ***Länge 15*** bearbeiten (siehe

dazu oben das Unterprogramm `put2` ab Zeile 47, den Aufruf von `put2` in Zeile 88, und das Beispielprogramm `Reihungen03.cpp`, hier nicht wiedergegebene).

Eigentlich sollte man innerhalb ***eines*** eckigen Klammerpaars immer nur ***einen*** Index angeben. Für die Angabe von z. B. ***zwei*** Indizes sollte man ***zwei*** eckige Klammerpaare benutzen, z. B. so: `r2[1][3]`. Allgemein gilt in C (und C++) allerdings: sind `A` und `B` zwei ***Ausdrücke***, dann ist `A, B` auch ein Ausdruck mit dem ***Wert*** von `B` und den Seiteneffekten von `A` und `B`. Deshalb darf man auch so etwas unsinniges wie in Zeile 92 (Ausgabe in Zeile 126) schreiben.

7.2 C-Strings (Reihungen von char, Adressen von char)

In C wurden ***Strings*** durch ***Reihungen*** mit `char`-Komponenten dargestellt, und diese Möglichkeit gibt es auch heute noch in C++. Das Problem mit der ***Länge*** von Reihungen löst man bei solchen ***C-Strings*** üblicherweise dadurch, dass man einen String ***mit einem Null-Byte abschliesst*** (Null-begrenzte Strings, null terminated strings). Ein solcher `char`-Wert `0` am Ende eines Strings sollte nicht mit dem `char`-Wert `'0'` (bei manchen C++-Ausführern identisch mit dem `char`-Wert `48`) oder einem Adresswert `NanA` (alias `NULL`, alias `0`) verwechselt werden.

Ein ***Null-begrenzter String*** hat konzeptuell ***zwei Längen***: Die ***physikalische Länge*** gibt an, wieviele Bytes im Speicher für ihn reserviert sind. Die ***logische Länge*** gibt an, wieviele Zeichen vor dem (hoffentlich vorhandenen) abschliessenden Null-Byte stehen.

Ein beliebter ***Programmierfehler*** besteht darin, die abschliessende `0` eines C-Strings versehentlich durch ein anderes Zeichen zu ersetzen. Zusammen mit der ***mangelnden Indexprüfung*** bei Reihungen bewirkt das praktisch eine Verlängerung des Strings bis zur nächsten `0` im Speicher.

Wenn irgend möglich sollte man anstelle von "alten C-Strings" lieber "moderne" Objekte der Klasse `string` verwenden (siehe Abschnitt 9.2). Allerdings ist man als C++-Programmierer häufig ***gezwungen***, mit C-Strings zu arbeiten.

```
1   // Datei CStrings01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert C-Strings (Vereinbarung, Initialisierung, Ausgabe) und die
4   Funktionen strcat, strcmp, strlen und sizeof.
5   ------------------------------------------------------------------------ */
6   #include <cstring>   // fuer strcpy, strcat, strcmp, strlen
7   #include <iostream>  // fuer cout
8   using namespace std;
9   // ------------------------------------------------------------------------
10  int main() {
11     cout << "CStrings01: Jetzt geht es los!"     << endl;
12
13     // C-String-Variablen vereinbaren:
14     char          t1[]   = "Hallo ";       // Reihung von char
15     char * const  t2     = "Susi!";        // Konstante Adresse von char
16     char          t3[20];                  // Reihung, nicht initialisiert
17     char          t4[]   = "Hallo Susi!";
18
19     // Einen C-String kopieren:
20     strcpy(t3, t1);                        // "Hallo " nach t3 kopieren
21     strcpy(t3, "Hallo ");                  // "Hallo " nach t3 kopieren
22
23     // C-Strings konkatenieren:
24     strcat(t3, "Susi!");                   // "Susi!" an t3 anhaengen
25
```

```
26     // C-Strings (mit Erlaeuterung davor) ausgeben:
27     cout << "t1:              " << t1             << endl;
28     cout << "t2:              " << t2             << endl;
29     cout << "t3:              " << t3             << endl;
30     cout << "t4:              " << t4             << endl;
31
32     // C-Strings vergleichen:
33     cout << "strcmp(t2, t1): " << strcmp(t2, t1) << endl;
34     cout << "strcmp(t1, t2): " << strcmp(t1, t2) << endl;
35     cout << "strcmp(t3, t4): " << strcmp(t3, t4) << endl;
36
37     cout << "'S' - 'H':       " << 'S' - 'H'      << endl;
38
39     // Die logische und die physikalische Laenge eines C-Strings:
40     cout << "strlen(t1):      " << strlen(t1)     << endl;
41     cout << "sizeof(t1):      " << sizeof(t1)     << endl;
42
43     cout << "strlen(t2):      " << strlen(t2)     << endl;
44     cout << "sizeof(t2):      " << sizeof(t2)     << endl;
45
46     cout << "strlen(t3):      " << strlen(t3)     << endl;
47     cout << "sizeof(t3):      " << sizeof(t3)     << endl;
48
49     // Auf einzelne Komponenten eines C-Strings zugreifen:
50     cout << "t3[0]:           " << t3[0]          << endl;
51     cout << "t3[6]:           " << t3[6]          << endl;
52
53     t3[ 1] = 'e';
54     t3[10] = '?';
55     cout << "t3:              " << t3             << endl;
56
57     cout << "CStrings01: Das war's erstmal!"     << endl;
58  } // main
59  /* ------------------------------------------------------------------------
60  Ausgabe des Programms CStrings01:
61
62  CStrings01: Jetzt geht es los!
63  t1:             Hallo
64  t2:             Susi!
65  t3:             Hallo Susi!
66  t4:             Hallo Susi!
67  strcmp(t2, t1): 11
68  strcmp(t1, t2): -11
69  strcmp(t3, t4): 0
70  'S' - 'H':      11
71  strlen(t1):     6
72  sizeof(t1):     7
73  strlen(t2):     5
74  sizeof(t2):     4
75  strlen(t3):     11
76  sizeof(t3):     20
77  t3[0]:          H
78  t3[6]:          S
79  t3:             Hello Susi?
80  CStrings01: Das war's erstmal!
81  --------------------------------------------------------------------- */
```

```
82  // Datei CStrings02.cpp
83  /* ------------------------------------------------------------------------
84  Die Funktion strlen funktioniert auch bei C-String-Parametern eines Unter-
85  programms wie erwartet, die Funktion sizeof aber nicht. In einem Unter-
86  programm ist die physikalische Laenge eines C-String-Parameters *nicht*
87  bekannt.
88  ------------------------------------------------------------------------ */
89  #include <cstring>   // fuer strlen
90  #include <iostream>  // fuer cout
91  using namespace std;
92  // ------------------------------------------------------------------------
93  void put(char p[]) {
94     // Gibt den C-String-Parameter p und "seine Laengen" (strlen und sizeof)
95     // aus:
96     cout << "p:         " << p         << endl;
97     cout << "strlen(p): " << strlen(p) << endl;
98     cout << "sizeof(p): " << sizeof(p) << endl;
99  } // put
100 // ------------------------------------------------------------------------
101 int main() {
102    // Eine C-String-Variable s vereinbaren und initialisieren:
103    char s[10] = "Hallo!";
104
105    // s und "seine Laengen" (strlen und sizeof) direkt ausgeben:
106    cout << "s:         " << s         << endl;
107    cout << "strlen(s): " << strlen(s) << endl;
108    cout << "sizeof(s): " << sizeof(s) << endl;
109
110    // s und "seine Laengen" mit Hilfe des Unterprogramms put ausgeben:
111    put(s);
112 } // main
113 /* ------------------------------------------------------------------------
114 Ausgabe des Programms CString02:
115
116 s:         Hallo!
117 strlen(s): 6
118 sizeof(s): 10
119 p:         Hallo!
120 strlen(p): 6
121 sizeof(p): 4
122 ------------------------------------------------------------------------ */
```

Die Programme `CString03` und `CString04` (hier nicht wiedergegeben) enthalten weitere Beispiele für die Benutzung alter C-Strings (Variablen der Typen `char[]` und `char*`).

7.3 Adresstypen und Referenztypen

Adresstypen (engl. pointer types) und ***Referenztypen*** (engl. reference types) gehören zu den konstruierten Typen, d. h. man kann sie nicht vereinbaren, sondern bekommt sie "zu jedem vereinbarten Typ geschenkt". Adresstypen gibt es auch in C, Referenztypen wurden erst in C++ eingeführt.

Zu den Referenztypen in C++ gibt es in Java keine Entsprechung. Den Adresstypen in C/C++ entsprechen (mehr oder weniger) die Referenztypen in Java.

Adresstypen (engl. pointer types) werden im Deutschen häufig auch als Pointertypen oder Zeigertypen bezeichnet. Das hat dann aber die sprachlich unschöne Konsequenz, dass "der Adressoperator & einen Pointer oder einen Zeiger liefert", statt (wie man von einem Adressoperator eigentlich erwartet) eine Adresse. Deshalb wird im Folgenden der englische Begriff ***pointer type*** immer mit ***Adresstyp*** und ***pointer*** mit ***Adresse*** übersetzt.

Weil Adresstypen und Referenztypen in C++ so wichtig sind und weil man viel darüber lernen kann, werden sie nicht in diesem Unterabschnitt, sondern im nächsten Hauptabschnitt behandelt. Hier folgt noch eine Liste wichtiger Grundbegriffe zusammen mit alternativen Bezeichnungen, die in der Literatur verwendet werden.

Hier verwendete Begriffe	***Alternative Bezeichnungen in anderen Schriften***
Adresstyp	Pointertyp, Zeigertyp, Referenztyp, pointer type
Referenztyp	Adresstyp, reference type, address type
Adressoperator &	address-of-operator
Variablenoperator *	Indirektionsoperator, Dereferenzierungsoperator, indirection operator

8 Variablen, Adresstypen und Referenztypen in C++

In C++ gibt es verschiedene Arten von Typen, darunter auch ***Adresstypen*** (pointer types, z. B. `int *`, `string *` und `int * *`) und ***Referenztypen*** (reference types, z. B. `int &` und `string &`). Diese Typen sollen hier genauer beschrieben werden. Als Grundlage werden vorher einige Fachbegriffe (Variable, unveränderbare Variable, Konstante etc.) eingeführt.

Zum Stoff dieses Kapitels siehe auch die Beispielprogramme in den Dateien `Adressen01.cpp` bis `Adressen25.cpp` und `Referenz01.cpp` bis `Referenz04.cpp`.

In den Beispielen der folgenden Abschnitte werden die folgenden Vereinbarungen vorausgesetzt:

```
1   // Variablen verschiedener Typen:
2   int    iv=171,      iva=251,      ivb=-31;
3   short  sv=17,       sva=25,       svb=-3;
4   double dv=17.5,     dva=25.5,     dvb=-3.5;
5   bool   bv=true,     bva=false,    bvb=true;
6   string tv="Hallo ", tva="Sonja!", tvb="Wie geht's?";
7
8   // Reihungen verschiedener Typen:
9   int    ir[] = {111,   222,      333,    444,    555};
10  short  sr[] = {11,    22,       33,     44,     55};
11  double dr[] = {11.1,  22.2,     33.3,   44.4,   55.5};
12  bool   br[] = {false, true,     true,   false,  true};
13  string tr[] = {"Hi ", "Sunny!", " How", " are", " you?"};
```

8.1 Variablen, unveränderbare Variablen und Konstanten

Def.: Eine ***Variable*** besteht im Wesentlichen aus zwei Teilen, einer ***Adresse*** und einem ***Wert***. Einige Variablen haben zusätzlich einen (oder mehrere) ***Namen*** und/oder einen ***Zielwert***.

Namen von Variablen werden vom Programmierer festgelegt, Adressen dagegen vom Ausführer. In den folgenden Beispielen werden die Adressen eines fiktiven, aber realistischen Ausführers gezeigt. Es handelt sich dabei stets um maximal vierstellige Zahlen im 16-er-System (Hexadezimalzahlen). Andere Ausführer legen für dieselben Variablen andere Adressen fest.

Beispiel-01: Variable und ihre Teile

Als Bojen dargestellt sehen die Variablen `iv`, `sv`, `dv`, `bv` und `tv` etwa so aus:

```
1   Name  Adresse  Wert
2   |iv|--<3000>--[171]
3   |sv|--<300c>--[17]
4   |dv|--<3018>--[17.5]
5   |bv|--<3030>--[true]
6   |tv|--<6014>--["Hallo "]
```

Die erste Boje (in Zeile 2) stellt eine Variable dar, die aus dem Namen `|iv|`, der Adresse `<3000>` und dem Wert `[171]` besteht. Für die anderen Bojen gilt Entsprechendes. In Darstellungen von Variablen als Bojen werden hier ***Namen*** immer in senkrechte Striche `|...|`, ***Adressen*** in spitze Klammern `<...>` und ***Werte*** in eckige Klammern `[...]` eingefasst.

Den Wert einer Variablen wie `iv` kann man z. B. mit einer Zuweisung ***verändern***. Vereinbart man eine Variable mit dem Typspezifizierer `const`, dann kann man ihren Wert (normalerweise) nicht verändern, und wir bezeichnen sie hier als ***unveränderbare Variable***.

Beispiel-02: Unveränderbare Variablen unterschiedlicher Typen

```
7   int    const uiv = 171;
8   short  const usv = 17;
9   double const udv = 17.5;
10  bool   const ubv = false;
11  string const utv = "Sonja!";
```

Damit C/C++-Programmierer sich nicht unnötig eingeschränk fühlen, darf man statt `int const` auch `const int` etc. schreiben (leider wird das ***Lesen*** von C/C++-Programmen durch solche Varianten der Notation nicht leichter).

Def.: Eine ***Konstante*** besteht ("nur") aus einem ***Namen*** und einem ***Wert***. Eine Konstante besitzt ***keine*** Adresse.

Der Zusammenhang zwischen dem Namen einer Konstanten und ihrem Wert muss normalerweise vom ***Programmierer*** durch eine entsprechende Vereinbarung hergestellt werden. Der Programmierer kann Fehler machen und einer Konstanten aus Versehen einen falschen Wert geben.

Anmerkung: Ein Literal wie etwa `123`, `0.5`, `'A'`, `"Hallo!"` etc. hat Ähnlichkeit mit einer Konstanten und ist auch so etwas wie ein Name, der für einen bestimmten Wert steht. Anders als bei Konstanten wird der Zusammenhang zwischen einem ***Literal*** und seinem ***Wert*** aber nicht vom Programmierer hergestellt, sondern durch (die Sprache und) den ***Ausführer*** festgelegt. Z. B. hat in Java das `float`-Literal `0.1F` immer den Wert `0.100000001490116119384765625` und das `char`-Literal `'A'` hat immer den Wert `65`. In C/C++ schreibt der Standard die Werte einiger Literale (z. B. `'A'`, `'B'`, ..., `0.1f`, `0.1` etc.) nicht verbindlich vor und unterschiedliche Ausführer verwenden unterschiedliche Werte. Glücklicherweise gibt es auch ein paar Gegenbeispiele. So haben z. B. die Literale `10`, `0XA`, `0xa` und `012` bei allen C/C++-Ausführern den Wert zehn.

In C/C++ kann man mit dem Präprozessor-Befehl `#define` Konstanten vereinbaren, z. B. so:

```
12  #define MWST 15.0
```

Manchmal ist es wichtig, solche "echten" Konstanten (die keine Adressen haben) von unveränderbaren Variablen wie `uiv`, `usv`, ... etc. (die Adressen besitzen) zu unterscheiden. In der Praxis (und an einigen Stellen dieses Skripts) werden unveränderbare Variable allerdings häufig auch als Konstanten bezeichnet (um Missverständnissen nicht völlig auszuschließen :-).

Nebenbei: Falls die Konstante `MWST` den in Deutschland gültigen Mehrwertsteuersatz darstellen soll, hat der Programmierer ihren Namen offenbar mit einem falschen Wert verbunden (und nach der Bundestagswahl im September 2005 wird der Fehler möglicherweise noch größer).

Außer `#define`-Konstanten gibt es in C/C++ noch andere Arten von Konstanten: ***Reihungen*** und ***Funktionen***. Der Name einer Reihung ist (in C/C++) technisch gesehen der Name einer ***Konstanten***, deren Wert eine ***Adresse*** ist.

Beispiel-03: Die Reihung (bzw. Adresskonstante) `ir` als Boje dargestellt

```
13  |ir  |--[<4034>]
14           <4034>--[111]
15           <4038>--[222]
16           <403c>--[333]
17           <4040>--[444]
18           <4044>--[555]
```

In Zeile 13 drücken die spitzen Klammern `<...>` aus, dass `4034` eine Adresse ist. Die zusätzlichen eckigen Klammern `[<...>]` sollen deutlich machen, dass die Adresse `<4034>` ein Wert ist, nämlich der Wert der Adresskonstanten `ir`.

Dieser Wert ist die Adresse der ersten Komponenten `ir[0]` der Reihung `ir`. Die zweite Komponente `ir[1]` hat in diesem Beispiel die Adresse `<4038>` und die letzte Komponente `ir[4]` die Adresse `<4044>`.

Dass `ir` eine ***Adresskonstante*** ist, hat zwei leicht überprüfbare Konsequenzen:

1. Wenn man `ir` ausgibt (z. B. mit dem Befehl `cout << ir << endl;`) so wird eine Adresse ausgegeben (z. B. `0x4034`), und nicht der `int`-Wert `111`, der an dieser Adresse steht oder der Name "ir" oder andere Daten.

2. Eine Zuweisung mit `ir` auf der linken Seite (z. B. `ir = ir+1;`) wird vom Ausführer abgelehnt (weil `ir` keine Variable sondern eine Konstante ist). Dagegen ist ein Ausdruck wie `ir+1` erlaubt und man kann seinen Wert z. B. mit dem Befehl `cout << ir+1;` ausgeben.

Auch der Name einer Funktion ist in C/C++ technisch der Name einer Adresskonstanten. Im Folgenden werden ***Adressen von Funktionen*** aber ***ignoriert*** und nur Adressen von (veränderbaren und unveränderbaren) Variablen betrachtet.

8.2 Adresstypen, Adressvariablen, Adresswerte und Adressen

AT-Regel: In C/C++ gibt es zu (fast) jedem Typ T einen sogenannten **Adresstyp** namens T * (lies: "Adresse einer T-Variablen" oder kurz: "Adresse von T"). Der Stern * ist hier ein Bestandteil des Typnamens (und nicht etwa ein Operator). Vor diesem Stern wird üblicherweise ein Blank oder kein Blank notiert, etwa so: T * oder T* (erlaubt sind vor dem Stern beliebig viele transparente Zeichen, d. h. Blanks, Tabzeichen oder Zeilenwechsel).

Beispiel-01: Adresstypen

Zum Typ int	gibt es den Adresstyp int *	("Adresse von int"),
zum Typ double	gibt es den Adresstyp double *	("Adresse von double") und
zum Typ string	gibt es den Adresstyp string *	("Adresse von string").

Die AT-Regel gilt insbesondere auch für ***Adresstypen*** (d. h. man darf die Regel rekursiv anwenden).

Beispiel-02: 2-Sterne-Adresstypen

Zum Typ int * gibt es den Adresstyp int * * ("Adresse von Adresse von int"), zum Typ double** gibt es den Adresstyp double*** ("Adresse von Adresse von Adress von double") und zum Typ string*** gibt es den Adresstyp string**** ... und so weiter.

Die AT-Regel gilt nicht für sogenannten **Referenztypen**, die im Abschnitt 8.6 behandelt werden. Zu einem Referenztyp wie int & gibt es keinen Adresstyp int & *. Statt dessen sollte man einfach den Adresstyp int * verwenden.

Def.: Eine **Adressvariable** ist eine Variable eines Adresstyps.

Beispiel-03: Eine Adressvariablen und ihre Boje

```
1   int * aiv  = new int(171);
```

Diese Vereinbarung bewirkt, dass der Ausführer **zwei** Variablen erzeugt, und zwar eine Adressvariable (vom Typ int*) namens aiv und eine int-Variable ohne Namen mit dem Anfangswert 17. Die Adressvariable aiv wird mit der Adresse der int-Variablen initialisiert. Als Bojen dargestellt sehen diese beiden Variablen etwa so aus:

```
2    Name   Adresse   Wert    Zielwert
3   |aiv |--<f284>--[<b3c>]--[171]
```

Die Adressvariable besteht aus dem Namen aiv, der Adresse <f284> und dem Wert [<b3c>]. Die int Variable (die durch den Befehl new int(171) erzeugt wurde) hat keinen Namen und besteht aus der Adresse <b3c> und dem Wert [171]. Die int-Variable <b3c>]--[171] bezeichnen wir auch als die ***Zielvariable*** (oder kürzer: als das ***Ziel***) der Adressvariablen |aiv|--<f284>--[<b3c>].

Def.: Den Wert einer Adressvariablen bezeichnen wir hier als ***Adresswert***.

Z. B. hat die Adressvariable aiv den Adresswert [<b3c>]. Die spitzen Klammern <...> kennzeichnen eine Adresse und die eckigen Klammern [...] darumherum drücken aus, dass diese Adresse ein Wert ist, nämlich der Wert der Adressvariablen aiv.

Eine Adressvariable vom Typ int* enthält entweder die Adresse einer int-Variablen oder einen ***NanA-Wert*** ("not an address"). Ein NanA-Wert ist zwar ein Adresswert, aber ***keine*** Adresse.

Zur Erinnerung: Eine Gleitpunktvariable enthält entweder eine Gleitpunktzahl oder einen ***NaN-Wert*** ("not a number"). Ein NaN-Wert ist zwar ein Gleitpunktwert, aber ***keine*** Gleitpunktzahl.

Anmerkung: NanA-Werte werden häufig als ***Null-Werte*** bezeichnet. In C/C++ wandelt der Ausführer den Ganzzahlwert `0` automatisch in einen `NanA`-Wert um, wenn es typenmäßig erforderlich ist. Deshalb kann man z. B. das Literal `0` benutzen, wenn man einen `NanA`-Wert angeben will.

Der C++-Standard legt nicht fest, wieviele `NanA`-Werte (alias Null-Werte, alias `NULL`, alias `0`) es gibt und durch welche Bitkombinationen sie dargestellt werden. Es muss aber gelten: Vergleicht man (mit der Operation `==`) einen `NanA`-Wert mit einer Adresse (bzw. mit einem anderen `NanA`-Wert) muss das Ergebnis gleich `false` (bzw. `true`) sein. Es ist deshalb üblich so zu sprechen und zu schreiben, als gäbe es nur ***einen*** NanA-Wert.

Anmerkung: Schade dass es nicht zu jedem Typ `T` mindestens einen Wert `NaTV` ("not a T-value") gibt. Das wäre besonders systematisch und leicht zu lernen.

Beispiel-04: Adressvariablen mit dem Adresswert `NanA` (alias `NULL`, alias `0`)

```
4   #define NanA 0
5   #define NULL 0
6
7   int * aiva = NanA;
8   int * aivb = NULL;
9   int * aivc = 0;
```

Technisch ist `NULL` ein Präprozessor-Makro, welches in der Kopfdateien `<cstddef>` definiert wird, etwa so wie in Zeile 5. Hier wird vorgeschlagen, statt dessen ein Makro namens `NanA` wie in Zeile 4 zu definieren und zum Initialisieren von Adressvariablen zu verwenden (um die Analogie zu den `NaN`-Werten für Gleitpunktvariablen zu betonen). Einem C/C++-Ausführer ist es gleichgültig, ob man zum Initialisieren einer Adressvariablen das Markro `NanA`, das Makro `NULL` oder das `int`-Literal `0` benutzt, das Ergebnis ist in allen drei Fällen gleich.

Als Bojen stellen wir Adressvariablen, deren Werte keine Adressen sind, immer so dar:

```
10  |aiva|--<ff00>--[NanA]
11  |aivb|--<ff04>--[NanA]
12  |aivc|--<ff08>--[NanA]
```

Beispiel-05: Adressen von Adressen von ...

```
13  int *     a1 =                            new int(171);
14  int * *   a2 =                new int * (new int(172));
15  int * * * a3 = new int * * (new int * (new int(173)));
```

Als Bojen dargestellt sehen diese Variablen etwa so aus:

```
16  |a1|--<1000>--[<A000>]--[171]
17  |a2|--<1004>--[<A004>]--[<B000>]--[172]
18  |a2|--<1008>--[<A008>]--[<B004>]--[<C000>]--[173]
```

Zusammenfassung: Einen Typ, dessen Name mit einem oder mehreren Sternchen endet, bezeichnen wir hier als einen **Adresstyp**, z. B. `int*`, `short**`, `string***` etc. Die Variablen solcher Typen bezeichnen wir als **Adressvariablen** und die Werte solcher Variablen als **Adresswerte**. Ein Adresswert ist entweder eine Adresse oder gleich `NanA` ("not an address"). Der Wert `NanA` ist ***keine*** Adresse und somit ist auch der Wert einer Adressvariablen nicht immer eine Adresse.

Adresstypen werden in der Literatur auch als ***Zeigertypen*** oder ***Pointertypen*** und im Englischen meist als ***pointer types*** bezeichnet.

8.3 R-Werte und L-Werte

Zur Erinnerung: ***Ausdrücke*** sind ***syntaktische*** Größen, die in einem Quellprogramm vorkommen können. ***Werte*** sind ***semantische*** Größen, die während der Ausführung eines Programms vom Ausführer berechnet, in Variablen gespeichert, als Parameter an Unterprogramme übergeben oder sonstwie bearbeitet werden. Ein Ausdruck bezeichnet einen Wert (oder: Ein Ausdruck befiehlt dem Ausführer, einen Wert zu berechnen).

Eine Variable besteht (mindestens) aus einer ***Adresse*** und einem ***Wert***. Manchmal bezeichnet man die Adresse auch als **L-Wert** und den Wert als **R-Wert** der Variablen.

Beispiel-01: L- und R-Werte von Variablen

```
1     Name   Adresse  Wert   <-- alte Bezeichnungen fuer die Teile einer Var.
2     Name   L-Wert   R-Wert <-- neue Bezeichnungen fuer die Teile einer Var.
3    |iv  |--<3000>--[171]
4    |iva |--<3200>--[251]
5    |ivb |--<3204>--[-31]
```

Die Variable `iv` hat hier den L-Wert `<3000>` und den R-Wert `[171]`.

Die Bezeichnungen **L-** und **R-** sollen an die **linke** bzw. **rechte** Seite einer Zuweisung erinnern. Die Zuweisung

```
6   iva = ivb;
```

kann man etwa so ins Deutsche übersetzen: "Speichere den ***Wert*** der Variablen `ivb` an die ***Adresse*** der Variablen `iva`". Diese Übersetzung soll deutlich machen: Von der Variablen `ivb` auf der ***rechten*** Seite der Zuweisung ist vor allem der R-Wert wichtig, von der Variablen `iva` auf der ***linken*** Seite wird nur der L-Wert (ihre Adresse) benötigt.

Werte werden in einem Quellprogramm immer durch ***Ausdrücke*** beschrieben. Ausdrücke, die L-Werte (bzw. R-Werte) beschreiben, bezeichnen wir hier als **L-Ausdrücke** (bzw. **R-Ausdrücke**). Der Name einer Variablen (z. B. `iv`) ist ein L-Ausdruck, Literale (z. B. `171` oder `'X'`) und die Namen (echter) Konstanten sind Beispiele für R-Ausdrücke.

In einem gewissen Sinn leistet ein L-Ausdruck ***mehr*** als ein R-Ausdruck. Ein L-Ausdruck bezeichnet direkt eine Adresse (einen L-Wert) und außerdem indirekt den Wert, der an dieser Adresse steht (den R-Wert des L-Ausdrucks). Ein L-Ausdruck bezeichnet also (direkt) einen L-Wert und (indirekt) einen R-Wert. Ein R-Ausdruck bezeichnet nur einen R-Wert. Man kann auch sagen: Ein L-Ausdruck bezeichnet eine ***Variable***. Ein R-Ausdruck bezeichnet nur einen ***Wert***.

Anmerkung: Man beachte, dass ein L-Ausdruck eine ***Adresse*** bezeichnet, und nie einen `NanA`-Wert.

LR-Regel: An allen Stellen eines Quellprogramms, an denen ein R-Ausdruck erwartet wird, darf man auch einen L-Ausdruck hinschreiben. Der Ausführer nimmt dann automatisch den R-Wert des L-Ausdrucks.

Beispiel-02: Ein L-Ausdruck als Bezeichnung für einen R-Wert

Auf der rechten Seite einer Zuweisung erwartet der Ausführer immer einen R-Ausdruck. Der Name einer Variablen, z. B. `ivb`, ist ein L-Ausdruck. Schreibt man `ivb` auf die rechte Seite einer Zuweisung, etwa so:

```
7   iva = ivb;
```

dann nimmt der Ausführer automatisch den R-Wert der Variablen, und nicht ihren L-Wert.

In den folgenden beiden Beispielen werden L-Ausdrücke immer links und R-Ausdrücke rechts von einem Zuweisungsoperator "=" notiert, um damit "ihren Charakter zu betonen".

Beispiel-03: L-Ausdrücke

Jeder einfache ***Variablenname*** ist ein L-Ausdruck:

```
8   iv=...; sv=...; dv=...; aiv=...;
```

Jeder Name einer ***Reihungskomponente*** ist ein L-Ausdruck:

```
9   ir[0]=...; ir[1]=...; ir[iv]=...; ir[2*iv-3]=...; dr[ir[2*iv-3]]=...;
```

Der dreistellige ***if-Operator*** `...?...:...` liefert einen L-Wert, wenn sein zweiter und sein dritter Operand L-Ausdrücke sind:

```
10  (0<iv && iv<=4 ? ir[iv] : ir[0]) = ...;
```

Verändert wird durch diese Zuweisung entweder die Variable `ir[iv]` oder die Variable `ir[0]`, je nachdem, ob der Ausdruck `0<iv && iv<=4` den Wert `true` oder `false` hat. Die runden Klammern um den gesamten L-Ausdruck sollen das Lesen erleichtern, können aber auch weggelassen werden.

Beispiel-04: R-Ausdrücke

Jedes **Literal** ist ein R-Ausdruck:

```
11  ...=17;   ...=3.5;   ...="Hallo!";
```

Die meisten mit ***Operatoren*** gebildeten Ausdrücke sind R-Ausdrücke:

```
12  ...=iv+1;    ...=2*sv+321;   ...=ir[0]+1;   ...=5*ir[2*iv-3]-1;
13  ...=iv++;    ...=++iv;       ...=iv--;      ...=--iv;
```

Eine erste Ausnahme von dieser Regel wurde oben in Zeile 10 vorgestellt. Eine besonders wichtige Ausnahme ist der Variablen-Operator `*`, der im nächsten Abschnitt behandelt wird.

Ist `f` eine Funktion mit einem "normalen" Rückgabetyp wie z. B. `int`, `double`, `string`, `int*`, `int**` ... etc., dann ist jeder Aufruf von `f` ein R-Ausdruck.

```
14  ...=sin(3.141);   ...=sqrt(2.0*cos(2.5));
```

Es gibt auch Funktionen, deren Aufrufe L-Ausdrücke sind. Solche Funktionen haben als Rückgabetyp einen sogenannten Referenztyp. Referenztypen werden im Abschnitt 8.6 behandelt.

Der Name einer Konstanten ist ein R-Ausdruck:

```
15  #define MWST 22.5   // Schon wieder ein falscher Mehrwertsteuersatz.
16  #define PI   3.141  // Ein schlechter Naeherungswert fuer Π.
17  ...=MWST;   ...=PI;
18  ...=ir;     ...=dr; // Der Name einer Reihung ist eine (Adress-) Konstante!
```

Ende von Beispiel-04: R-Ausdrücke

Achtung: Es gibt L-Ausdrücke die ***nicht*** auf der linken Seite einer Zuweisung stehen dürfen. Der Name ***jeder*** Variablen ist ein L-Ausdruck, aber einer unveränderbaren Variablen (wie z. B. `uiv`, `usv`, ... etc.) darf man nichts zuweisen:

```
19  uiv=...; // Verboten, weil uiv mit "const" vereinbart wurde!
```

Trotzdem gilt der Name einer unveränderbaren Variablen wie `uiv` als L-Ausdruck, weil auch `uiv` eine Adresse (einen L-Wert) und einen Wert (einen R-Wert) hat.

8.4 Der Variablen-Operator * und der Adress-Operator &

Wie ihre Namen schon vermuten lassen, haben der Variablen-Operator `*` und der Adress-Operator `&` mit Variablen und Adressen zu tun:

Der Adress-Operator & bildet eine **Variable** auf ihre **Adresse** ab.

Der Variablen-Operator * bildet eine **Adresse** auf ihre **Variable** ab.

Die folgende Darstellung soll diese kurzen und unvollständigen (aber hoffentlich "leicht merkbaren") Erläuterungen "graphisch ergänzen":

```
  Variable      Operator    Adresse
<abc>--[def]    --- & -->    <abc>
<abc>--[def]    <-- * ---    <abc>
```

Die Operatoren `*` und `&` haben auch mit L- und R-Ausdrücken zu tun:

Der Adress-Operator & bildet einen L-Wert (eine Variable) auf einen R-Wert (die Adresse der Variablen) ab.

Der Variablen-Operator * bildet einen R-Wert (die Adresse einer Variablen) auf einen L-Wert (die Variable) ab.

Nach diesen Vorbereitungen folgt hier eine vollständige Beschreibung des Adressoperators &:

Sei `LA` ein beliebiger L-Ausdruck, der eine Variable `V` bezeichnet. Dann ist `&LA` ein R-Ausdruck, der die Adresse von `V` bezeichnet.

Beispiel-01: Den Adressoperator & auf Variablen anwenden

Der Ausdruck `&iv` bezeichnet die Adresse der Variablen `iv`. Der Ausdruck `&ir[0]` bezeichnet die Adresse der Variablen `ir[0]`. Der Ausdruck `&dr[ir[2*iv-3]]` bezeichnet die Adresse der Variablen `dr[ir[2*iv-3]]`.

"Normale" Ausdrücke dienen dazu, "normale Werte" zu bezeichnen (z. B. `int`-Werte, `double`-Werte oder `string`-Werte). Adressausdrücke dienen dazu, Adresswerte zu bezeichnen.

Beispiel-02: Variablen mit Ausdrücken initialisieren, die Adresswerte bezeichnen

```
1   int   iv   = 171;
2   int * aiva = &iv;            // Die Adresse der Variablen iv
3   int * aivb = new int(-31);   // Die Adresse einer neuen int-Variablen
4   int * aivc = NanA;           // Not an Address (aber ein Adresswert!)
```

Zur Veranschaulichung folgen hier die Bojen der Variablen `iv`, `aiva` , `aivb` und `aivc`:

```
5    Name   Adresse  Wert
6   |iv  |--<5000>--[171]
7                     |
8                     |
9   |aiva|--<ef04>--[<5000>]
10  |aivb|--<ef00>--[< a68>]--[-31]
11  |aivc|--<eefc>--[<NanA>]
```

Der Wert des Ausdrucks `&iv` ist ein Adresswert (nämlich die Adresse `<5000>` der Variablen `iv`). Die Adressvariable `aiva` wird mit der Adresse `&iv` initialisiert. Die Adressvariable `aivb` wird mit der Adresse `<a68>` einer mit `new` erzeugten `int`-Variablen initialisiert. Die Adressvariable `aivc` wird mit dem Adresswert `NanA` initialisiert. Wenn man die Variablennamen `aiva`, `aivb` und `aivc` auf der rechten Seite einer Zuweisung verwendet, bezeichnen sie somit die Adresswerte `<5000>`, `<a68>` bzw. `NanA`.

Es folgt eine (fast) vollständige Beschreibung des Variablen-Operators *:

Sei RA ein beliebiger R-Ausdruck, der die Adresse einer Variablen V bezeichnet. Dann ist *RA ein L-Ausdruck, der die ("ganze") Variable V bezeichnet.

Beispiel-03: Den Variablenoperator * auf Adressen anwenden

Die Adressvariable aiva hat den Wert <5000>. Der Ausdruck *aiva bezeichnet die Variable <5000>--[171] (d. h. die Variable iv). Die Adressvariable aivb hat den Wert <a68>. Der Ausdruck *aivb bezeichnet die Variable <a68>--[-31] (diese Variable hat keinen Namen). Die Adressvariable aivc hat den Wert <NanA>. Deshalb bezeichnet der Ausdruck *aivc keine Variable, sondern löst eine ***Ausnahme*** aus. Der Ausdruck *&iv bezeichnet die Variable iv.

Beispiel-04: Werte vertauschen mit Adresstypen

Angenommen, wir haben zwei Variablen des selben Typs, etwa so:

```
12  double d1 = 1.7;
13  double d2 = 2.5;
```

und wollen mit Hilfe eines Unterprogramms die Werte der Variablen vertauschen, etwa so:

```
14  vertausche(...d1..., ...d2...);
```

Die Auslassungen ... sollen andeuten, dass dieser Unterprogrammaufruf nur ein erster Plan ist und noch verändert werden muss (siehe unten Zeile 20). Wir möchten, dass nach Ausführung des Aufrufs die Variable d2 den Wert 1.7 und die Variable d1 den Wert 2.5 hat.

Wenn wir das Unterprogramm vertausche mit zwei double-Parametern schreiben, dann kann es nicht richtig funktionieren. Denn es bekommt dann ***Kopien*** der Werte der aktuellen Parameter d1 und d2, und egal was das Unterprogramm mit diesen Kopien macht, bleiben die Variablen d1 und d2 unverändert.

Der entscheidende Trick besteht darin, dem Unterprogramm nicht die double-Werte der Variablen zu übergeben, sondern ihre ***Adressen*** (vom Typ double *), etwa so:

```
15  void vertausche(double * a1, double * a2) {
16     double tmp = *a1;
17     *a1        = *a2;
18     *a2        = tmp;
19  }
```

Man beachte, dass die Variable tmp als double-Variable (und nicht als double*-Variable) vereinbart wurde, weil sie einen double-Wert aufnehmen soll und keinen Adresswert. Ein Aufruf dieses Unterprogramms kann dann etwa so aussehen:

```
20  vertausche(&d1, &d2);
```

Die Ausführung dieses Aufrufs beginnt damit, dass die formalen Parameter a1 und a2 als Variablen erzeugt und mit den aktuellen Parametern &d1 und &d2 initialisiert werden, etwa so:

```
21  double * a1 = &d1;
22  double * a2 = &d2;
```

Danach sehen die Variablen d1, d2, a1 und a2 als Bojen dargestellt etwa so aus:

```
23  |d1|--<A010>---[1.7]
24                   |
25  |a1|--<B000>--[<A010>]
26
27  |d2|--<A014>---[2.5]
28                   |
29  |a2|--<B004>--[<A014>]
```

Die L-Ausdrücke `*a1` und `*a2` bezeichnen die Variable `d1` bzw. `d2`. Über die L-Ausdrücke hat das Unterprogramm Zugriff auf diese Variablen (nicht nur auf ihre Werte!) und kann ihre Werte vertauschen.

Beispiel-05: L- und R-Ausdrücke sind auch dann verschieden, wenn sie gleiche Werte haben

```
30  int   n = 171;
31  int * a = &n;
```

Als Bojen dargestellt sehen die Variablen n und a etwa so aus:

```
32   Name    L-Wert   R-Wert
33  |n   |--<B000>---[171]
34                     |
35  |a   |--<C000>--[<B000>]
```

Jetzt gilt:

Der L-Ausdruck `n` hat den L-Wert `<B000>` (und den R-Wert `[171]`)
Der L-Ausdruck `a` hat den R-Wert `<B000>` (und den L-Wert `<C000>`)
Der R-Ausdruck `&n` hat den R-Wert `<B000>`

Obwohl der L-Ausdruck `n` und der R-Ausdruck `&n` beide den Wert `<B000>` haben, darf man nur `n` auf die linke Seite einer Zuweisung schreiben, aber nicht `&n`. Der L-Wert `<B000>` ist eben etwas anderes als der R-Wert `<B000>`.

Beispiel-06: Adressen von Adressen von ...

```
36  int       v0 = 171;
37  int *     v1 = &v0;
38  int * *   v2 = &v1;
39  int * * * v3 = &v2;
```

Als Bojen sehen diese Variablen etwa so aus:

```
40  |v0|--<4000>---[171]
41                   |
42  |v1|--<5000>--[<4000>]
43                   |
44  |v2|--<6000>--[<5000>]
45                   |
46  |v3|--<7000>--[<6000>]
```

Die Variable `v3` ist vom Typ ***Adresse von Adresse von Adresse von*** `int` und hat hier den Wert `<6000>`. Fuer die anderen Variablen gilt Entsprechendes.

Die Bezeichnung **Adressoperator** (engl. address-of operator) für den Operator & ist in der Literatur weit verbreitet. Die Bezeichnung **Variablenoperator** für den Operator * ist dagegen unüblich. Verbreitet sind die Bezeichnungen **Derefenzierungsoperator** und **Indirektionsoperator**. (engl. dereference operator, indirection operator). Diese Bezeichnungen machen aber nicht deutlich, dass der Operator eine Variable (oder: einen L-Wert) als Ergebnis liefert. Außerdem sind diese Bezeichnungen ziemlich lang und klingen komplizierter als **Variablenoperator**.

8.5 Mit Adresswerten rechnen

In C/C++ ist es dem Programmierer möglich, mit Adresswerten einfache Berechnungen (Additionen und Subtraktionen) durchzuführen. Z. B. darf er zwei Adressen (von Komponenten derselben Reihung) subtrahieren und er darf eine Ganzzahl zu einer Adresse addieren oder von einer Adresse subtrahieren.

Seien `AVA` und `AVB` zwei Ausdrücke eines Adresstyps `T *` und sei `GGG` ein Ausdruck eines Ganzzahltyps (wie `int`, `short`, `long`, `unsigned int`, ...). Dann sind folgende Ausdrücke erlaubt (und bezeichnen Werte der angegebenen Typen) bzw. verboten:

```
1   Ausdruck      Typ
2   AVA + AVB     verboten  // Adresse  plus  Adresse  verboten
3   AVA + GGG     T *       // Adresse  plus  Ganzzahl erlaubt
4   AVA - AVB     ptrdiff_t // Adresse  minus Adresse  erlaubt
5   AVA - GGG     T *       // Adresse  minus Ganzzahl erlaubt
6   GGG + AVA     T *       // Ganzzahl plus  Adresse  erlaubt
7   GGG - AVA     verboten  // Ganzzahl minus Adresse  verboten
```

Der Typname `ptrdiff_t` wird in der Kopfdatei `<cstddef>` definiert und bezeichnet einen vorzeichenbehafteten Ganzzahltyp (z. B. den Typ `int`).

Die Operationen Multiplizieren, Dividieren und Modulieren ("die Modulo-Operation % anwenden") sind für Adresswerte grundsätzlich ***nicht*** erlaubt.

Beispiel-01: Die Typen von Ausdrücken, in denen mit Adresswerten gerechnet wird

Den Operator `typeid` darf man auf einen beliebigen Ausdruck `A` anwenden. Als Ergebnis liefert er ein Objekt der Klasse `type_info` (definiert in der Kopfdatei `<typeinfo>`). Dieses Objekt enthält unter anderem eine Funktion namens `name`. Ein Ausdruck wie `typeid(A).name()` liefert den Namen des Typs von `A`, etwa so:

```
8   typeid(adva).name():      double *
9   typeid(advb).name():      double *
10  typeid(adva+1234).name(): double *
11  typeid(adva-advb).name(): int         // ptrdiff_t
12  typeid(adva-1234).name(): double *
13  typeid(1234+adva).name(): double *
```

Den Zeilen 8 und 9 kann man entnehmen, dass `adva` und `advb` Variablen des Typs `double *` sind. Die übrigen Zeilen zeigen die Typen von Ausdrücken, in denen diese Variablen vorkommen.

Anmerkung: Die Typnamen, die von einem Funktionsaufruf wie `typeid(adva).name()` geliefert werden, sind nicht standardisiert. Benützt man den Gnu-C++-Compiler gcc so liefert die Funktion `name` z. B. `"Pd"` anstelle von `"double *"` und `"i"` anstelle von `"int"` etc.

Sei `AT` ein Ausdruck des Typs `T *` (d. h. der Wert dieses Ausdrucks ist die Adresse einer T-Variablen oder gleich `NanA`). Der Wert des Ausdrucks `AT+1` ist dann nicht um 1 grösser als der Wert von `AT`, sondern um `sizeof(T)`, d. h. um die Länge einer T-Variablen, gemessen in Bytes.

Beispiel-02: Das kleine 1+1 für Adresswerte (multiplizieren darf man sie ja nicht)

```
14  typeid(aiv).name():       int *
15  typeid(asv).name():       short *
16  typeid(adv).name():       double *
17  typeid(abv).name():       bool *
18  typeid(atv).name():       string *
19
```

```
20  aiv: 1000, aiv+1: 1004, aiv+2: 1008, aiv+3: 100c
21  asv: 2000, asv+1: 2002, asv+2: 2004, asv+3: 2006
22  adv: 3000, adv+1: 3008, adv+2: 3010, adv+3: 3018
23  abv: 4000, abv+1: 4001, abv+2: 4002, abv+3: 4003
24  atv: 5000, atv+1: 5010, atv+2: 5020, atv+3: 5030
```

Den Zeilen 14 bis 18 kann man die Typen der (Adress-) Variablen `aiv`, `asv`, ... etc. entnehmen. In den Zeilen 20 bis 24 sieht man, dass die Adressvariablen erfreulich "glatte Werte" (`1000`, `2000` etc.) haben. Es sollte deshalb nicht allzu schwer sein nachzurechnen, dass (bei dem verwendeten Ausführer) eine `int`-Variable 4 Bytes, eine `short`-Variable 2 Bytes etc. belegt.

Auf die Komponenten einer Reihung `dra` greift man normalerweise mit Hilfe von Indizes und (L-) Ausdrücken wie `dra[3]`, `dra[i]` etc. zu. Statt mit einer Indexvariablen wie `i` kann man aber auch mit einer ***Adressvariablen*** auf die Reihungskomponenten zugreifen.

Beispiel-03: Auf die Komponenten einer Reihung mit Adressen statt mit Indizes zugreifen

```
25  double    dra[] = {1.11, 2.22, 3.33, 4.44, 5.55}; // Eine Reihung und
26  int const LEN   = sizeof(dra)/sizeof(dra[0]);     // ihre Laenge
27
28  for (double * a=dra; a<dra+LEN; a++) {
29     cout << "a: <" << hex << ai << ">, *a: " <<  *a << endl;
30  }
```

Hier ist die Adressvariable `a` ein Ersatz für die sonst übliche Indexvariable `i`, und anstelle des Ausdrucks `dra[i]` wird hier der Ausdruck `*a` verwendet, um auf die Komponenten der Reihung zuzugreifen. Man beachte, dass der Ausdruck `a++` den Wert der Adressvariablen `a` nicht um `1` erhöht, sondern um die Länge einer `double`-Variablen, weil `a` vom Typ `double *` ist.

Die Reihung `dra` und ihre Komponenten sehen als Bojen dargestellt etwa so aus:

```
31  |dra |--[<eec8>]
32           <eec8>--[1.11]
33           <eed0>--[2.22]
34           <eed8>--[3.33]
35           <eee0>--[4.44]
36           <eee8>--[5.55]
```

Die Ausgabe der obigen for-Schleife sieht dann etwa so aus:

```
37  a: eec8, *a: 1.11
38  a: eed0, *a: 2.22
39  a: eed8, *a: 3.33
40  a: eee0, *a: 4.44
41  a: eee8, *a: 5.55
```

Wann und warum sollte man mit einer Adressvariablen `a` statt mit einer Indexvariablen `i` auf die Komponenten einer Reihung zugreifen? Einige alte C-Programmierer können schneller einen Ausdruck wie `*a` eintippen, als einen Ausdruck wie `dra[i]`, in dem eckige Klammern vorkommen. Andere weisen darauf hin, dass ein Zugriff mit einer Adressvariablen (auf heute üblichen maschinellen Ausführern) etwas schneller geht als ein Zugriff mit einer Indexvariablen. Allerdings ist dieser Geschwindigkeitsunterschied nur in relativ seltenen Fällen wichtig und in sehr vielen Fällen spielt er überhaupt keine Rolle.

Auf jeden Fall muss jeder C/C++-Programmierer Schleifen wie die im Beispiel-03 lesen und verstehen können, weil sie oft in (den von den alten C-Programmierern geschriebenen) Programmen vorkommen.

Anmerkung: Im C-Standard (BS ISO/IEC 9899:1999, Abschnitt 6.5.2.1 Array subscripting) wird die Bedeutung der eckigen Klammern wie folgt definiert:

"The definition of the subscript operator [] is that `E1[E2]` is identical to `(*((E1)+(E2)))`."

Im C++Standard (BS ISO/IEC 14882:2003, Abschnitt 5.2.1 Subscripting) findet man eine ähnliche Definition. Sie hat folgende merkwürdige Konsequenz: Ist r eine Reihung und i eine int-Variable, dann haben die L-Ausdrücke `r[i]`, `(*(r+i))`, `(*(i+r))` und `i[r]` alle dieselbe Bedeutung. Welchen davon man verwendet ist Geschmacksache (um `i[r]` statt `r[i]` zu verwenden muss man aber einen ziemlich sonderbaren Geschmack haben :-).

Eine ***Adressvariable*** und ihre ***Zielvariable*** können unabhängig voneinander ***unveränderbar*** (konstant) sein oder nicht. Entsprechend gibt es zu jedem Typ T eigentlich ***vier*** Adresstypen, in deren Namen der Typqualifizierer `const` unterschiedlich häufig (null bis zweimal) und an unterschiedlichen Orten vorkommt, wie das folgende Beispiel verdeutlichen soll:

Beispiel-05: (un)veränderbare Adressvariablen und (un)veränderbare Zielvariablen

```
42  int ir[4] = {17, 27, 37, 47};
43                                   // Adressvariable:   Zielvariable:
44  int       *       ai01 = &ir[0]; // veraenderbar      veraenderbar
45  int const *       ai02 = &ir[2]; // veraenderbar      unveraenderbar
46  int       * const ai03 = &ir[1]; // unveraenderbar    veraenderbar
47  int const * const ai04 = &ir[3]; // unveraenderbar    unveraenderbar
48
49   *(ai01 ++);  //   Veraenderbare Adressvariable, erlaubt
50  (* ai01)++ ;  //   Veraenderbare   Zielvariable, erlaubt
51   *(ai02 ++);  //   Veraenderbare Adressvariable, erlaubt
52  (* ai02)++ ;  // Unveraenderbare   Zielvariable, verboten
53   *(ai03 ++);  // Unveraenderbare Adressvariable, verboten
54  (* ai03)++ ;  //   Veraenderbare   Zielvariable, erlaubt
55   *(ai04 ++);  // Unveraenderbare Adressvariable, verboten
56  (* ai04)++ ;  // Unveraenderbare   Zielvariable, verboten
```

Die Namen von Adresstypen liest man üblicherweise von rechts nach links (wie normales Arabisch oder Hebräisch), den Typenamen `int * const` etwa als "konstante Adresse von int" und den Typnamen `int const *` als "Adresse von unveränderbarer int-Variablen" oder kürzer als "Adresse von konst int".

Regel: Die Zielvariable einer Adressvariablen vom Typ `float const *` ("Adr. einer ***unveränderbaren*** float-Variablen") muss nicht unbedingt eine ***unveränderbare*** Variable sein, sondern darf auch eine ***veränderbare*** Variable sein.

Diese Regel mag im ersten Moment widersinnig erscheinen. Das folgende Beispiel soll aber deutlich machen, dass sie in Wirklichkeit ("ab dem zweiten oder dritten Moment") durchaus sinnvoll ist.

Beispiel-06: Adressvariablen mit und ohne "Veränderungsberechtigung"

```
57     float          vf    = 1.2; //   Veraenderbare float-Variable
58     float const    uf    = 3.4; // Unveraenderbare float-Variable
59
60     float const * auf01 = &vf; // Adr. einer unveraenderbaren float-Var.
61     float const * auf02 = &uf; // Adr. einer unveraenderbaren float-Var.
62
63     float       * avf01 = &vf; // Adr. einer   veraenderbaren float-Var.
64  // float       * avf02 = &uf; // Adr. einer   veraenderbaren float-Var.
```

Die Variable `vf` ist gleichzeitig das Ziel der Adressvariablen `auf01` und der Adressvariablen `avf01`, d. h. die drei Ausdrücke `vf`, `*auf01` und `*avf01` bezeichnen dieselbe `float`-Variable `<F000>--[1.2]`, wie die folgende Bojendarstellung verdeutlichen soll:

```
65  |  vf  |--<F000>----------[1.2]
66                              |
67  |auf01|--<F004>--[<F000>]--+
68                              |
69  |avf01|--<F008>--[<F000>]--+
```

Obwohl sie dieselbe Variable bezeichnen, sind die drei Ausdrücke `vf`, `*auf01` und `*avf01` aber nicht völlig "gleichwertig": `vf` und `*avf01` sind "mit einer ***Veränderungsberechtigung*** verbunden", der Ausdruck `*auf01` dagegen nicht (wegen dem Wort `const` in Zeile 60). Von den folgenden drei Anweisungen ("Veränderungsbefehlen") sind deshalb nur zwei erlaubt:

```
70        vf    =    vf    + 0.1; // Erlaubt
71      *auf01 = *auf01 + 0.1; // Verboten
72      *avf01 = *avf01 + 0.1; // Erlaubt
```

Man sagt auch: Die Adressvariable `auf01` bietet eine **unveränderbare Ansicht** (a constant view) der Variablen `vf`, `avf01` bietet dagegen eine **veränderbare Ansicht** (a non-constant view) der selben Variablen.

Aufgabe 1: Begründen Sie, warum die Vereinbarung der Adressvariablen `avf02` (oben in Zeile 64) verboten ist (und deshalb "auskommentiert" wurde). Was für eine Ansicht (eine veränderbare oder eine unveränderbare?) würde die Adressvariable `avf02` auf was für eine Variable (eine veränderbare oder eine unveränderbare?) bieten?

Aufgabe 2: Sind die folgenden Anweisungen ("Veränderungsbefehle") ***erlaubt*** oder ***verboten***?

```
73        uf    =    uf    + 0.1;
74      *auf02 = *auf02 + 0.1;
```

8.6 Referenztypen

RT-Regel: Zu (fast) jedem Typ `T` gibt es in C++ einen **Referenztyp** `T &` (lies: "Referenz auf T").

Beispiel-01: Referenztypen

Zum Typ `int` gibt es den Referenztyp `int &` ("Referenz auf int"), zum Typ `string` gibt es den Referenztyp `string &` ("Referenz auf string") u.s.w.

Die RT-Regel gilt **nicht** für Referenztypen (d. h. man darf die Regel nicht rekursiv anwenden). Zu einem Referenztyp `T &` gibt es keinen Typ `T & &`. Die RT-Regel gilt aber für alle anderen Typen, z. B. auch für Adresstypen.

Beispiel-02: Weitere Referenztypen

Zum Adresstyp `int *` gibt es den Referenztyp `int * &` ("Referenz auf Adresse von `int`"), zum Adresstyp `string*` gibt es den Referenztyp `string*&` ("Ref. auf Adr. von `string`") etc.

Referenztypen sind keine Baupläne für Variablen. Mit einer Variablenvereinbarung, die mit einem Referenztyp beginnt, etwa so:

```
1   int & n2 = ...
```

befiehlt man dem Ausführer nicht, eine weitere Variable zu erzeugen. Vielmehr wird einer schon vorhandenen Variablen der angegebene Name (im Beispiel: `n2`) als zusätzlicher Name zugeordnet. Die schon vorhandene Variable muss anstelle der Auslassung `...` angegeben werden (und muss im Beispiel zum Typ `int` gehören).

Beispiel-03: Eine Variable mit drei Namen vereinbaren

```
2   int   n1 = 17; // "Erzeuge eine int-Var. namens n1 mit dem Anfangswert 17"
3   int & n2 = n1; // "Gib der Variablen n1 den zusätzlichen Namen n2"
4   int & n3 = n1; // "Gib der Variablen n1 den zusätzlichen Namen n3"
```

Aufgrund dieser ***drei*** Vereinbarungen wird ***eine*** int-Variable mit den ***drei*** Namen `n1`, `n2` und `n3` und dem Anfangswert `17` erzeugt. Als Boje kann man diese Variable etwa so darstellen:

```
5   |n1|-+-<A000>--[17]
6        |
7   |n2|-+
8        |
9   |n3|-+
```

Welchen der drei Namen man benützt, um auf die Variable zuzugreifen, ist dem Ausführer gleichgültig. Nach den folgenden drei Zuweisungen

```
10  n3 = n1 + 3;
11  n2 = n2 + 2;
12  n1 = n3 + 1;
```

hat die Variable den Wert 23.

Obwohl durch eine Vereinbarung wie `int & n2 = n1;` keine neue Variable erzeugt wird, beschreibt man ihre Wirkung doch häufig so: Diese Vereinbarung ***erzeugt eine Referenzvariable*** namens `n2` (mit der Adresse und dem Wert der Variablen `n1`). Oder man sagt: Diese Vereinbarung führt den Namen `n2` als ***Alias*** (-Namen) für die Variable `n1` ein.

Das Beispiel-03 sollte die grundlegenden Regeln für Referenztypen verdeutlichen. In "erntshaften" C++-Programmen werden Referenztypen fast nie wie in diesem Beispiel verwendet, sondern fast ausschliesslich als ***Typen von Parametern*** oder als ***Rückgabetypen*** von Funktionen. Das folgende Beispiel ist praxisnäher als das vorige.

Beispiel-04: Werte von Variablen vertauschen mit Referenztypen

Wir haben wieder zwei Variablen des selben Typs, und wollen mit Hilfe eines Unterprogramms die Werte der Variablen vertauschen, etwa so:

```
13 double d1 = 1.7;
14 double d2 = 2.5;
15 vertausche(d1, d2);
```

Dieser Aufruf funktioniert wie gewünscht, wenn wir `vertausche` wie folgt vereinbaren:

```
16 void vertausche(double & al1, double & al2) {
17    double tmp = al1;
18    al1        = al2;
19    al2        = tmp;
20 }
```

Die Ausführung des Aufrufs `vertausche(d1, d2)` beginnt damit, dass die formalen Parameter `al1` und `al2` als Variablen erzeugt und mit den aktuellen Parameter `d1` un `d2` initialisiert werden, etwa so:

```
21 double & al1 = d1;
22 double & al2 = d2;
```

Dadurch wird `al1` (bzw. `al2`) zu einem Alias-Namen für `d1` (bzw. `d2`), was in Bojendarstellung etwa so aussieht:

```
23 |d1|-+-<A010>---[1.7]
24      |
25 |a1|-+
26
27 |d2|-+-<A014>---[2.5]
28      |
29 |a2|-+
```

Damit haben wir zwei verschiedene Lösungen für das Vertausche-Problem, eine mit ***Adresstypen*** und eine mit ***Referenztypen***. Es ist nützlich, diese Lösungen zu vergleichen.

Viele C++-Programmierer ziehen die zweite Lösung (die mit den Referenztypen) vor, weil sie im Aufruf ohne den Adressoperator & und im Rumpf des Unterprogramms ohne den Variablen-Operator * auskommt. Aber das ist kein sehr tiefgehender Unterschied sondern mehr eine Frage der Eleganz. Auch im Zeitbedarf dürften die beiden Lösungen sich bei vielen Ausführern kaum unterscheiden. Viele praxisnahe Probleme lassen sich wahlweise mit ***Adresstypen*** oder mit ***Referenztypen*** lösen, wobei die Lösung mit Referenztypen meist ein bisschen eleganter ist.

Nur wenn man selbst ***Operatoren*** (wie + oder * etc.) programmiert gibt es Fälle, die man nur mit Referenztypen (aber nicht mit Adresstypen) "gut lösen" kann. Diese Fälle werden hier aber nicht behandelt (siehe dazu das Buch "C++-Primer" von Lippman und Lajoie)..

Beispiel-05: Eine Funktion mit einem Referenztyp als Rückgabetyp

```
30 double & dv(int index) {
31    static double      sonder = 9.0;
32    static double      dr[5]  = {0.5, 1.5, 2.5, 3.5, 4.5};
33    static const  int LEN    = sizeof(dr)/sizeof(dr[0]);
34
35    if (0 <= index && index < LEN) {
36       return dr[index];
37    } else {
38       return sonder;
39    }
40 } // dv
```

Dieses Unterprogramm erwartet als Parameter einen Index und liefert die entsprechende Komponente einer Reihung `dr` als Variable. Falls der Index außerhalb der erlaubten Grenzen (0 bis 4) liegt, liefert das Unterprogramm die Variable `sonder` als Ergebnis. Die Reihung `dr` und die Variable `sonder` sind als lokale `static` Größen des Unterprogramms vereinbart, werden also nur ***einmal*** (beim ersten Aufruf von `dv`) erzeugt und leben dann "ewig" (d. h. bis das umgebende Programm sich endgültig beendet.

Es folgen zwei typische Aufrufe des Unterprogramms `dv`:

```
41     double & d1 = dv(2);
42     double & d2 = dv(7);
43
44     cout << setprecision(2) << showpoint;
45     cout << "A d1: " << d1 << ", d2: " << d2 << endl;
46     d1 = d1 + 0.3;
47     d2 = d2 + 0.4;
48     cout << "B d1: " << d1 << ", d2: " << d2 << endl;
```

In Zeile 41 und 42 werden nicht etwa zwei neue Variablen erzeugt, sondern nur Alias-Namen für schon existierende Variablen vergeben, die vom Unterprogramm `dv` geliefert werden. Der erste Aufruf `dv(2)` liefert die Variable `dr[2]` (diese Variable bekommt den Alias-Namen `d1`). Der zweite Aufruf `dv(7)` liefert die Variable `sonder` (diese Variable bekommt den Alias-Namen `d2`).

Die Befehle in den Zeilen 44 bis 48 erzeugen folgende Ausgabe:

```
49  A d1: 2.5, d2: 9.0
50  B d1: 2.8, d2: 9.4
```

Ende von Beispiel-05.

Auch ein Alias-Name kann (ganz ähnlich wie eine Adressvariable) mit einer "Veränderungsberechtigung" verbunden sein oder auch nicht, wie das nächste Beispiel verdeutlichen soll:

Beispiel-06: Referenzen auf `float` und Referenzen auf `const float`

```
51     float          vf    = 1.2; //   Veraenderbare float-Variable
52     float const    uf    = 3.4; // Unveraenderbare float-Variable
53
54     float const & ruf01 = vf;  // Alias fuer vf ohne Veraenderungsberecht.
55     float const & ruf02 = uf;  // Alias fuer uf ohne Veraenderungsberecht.
56
57     float       & rvf01 = vf;  // Alias fuer vf mit  Veraenderungsberecht.
58  // float const & rvf02 = uf;  // Alias fuer uf mit  Veraenderungsberecht.
59
60       vf   =  vf   + 0.1;      // Erlaubt
61  //  ruf01 = ruf01 + 0.1;      // Verboten
62      rvf01 = rvf01 + 0.1;      // Erlaubt
63
64  //   uf   =  uf   + 0.1;      // Verboten
65  //  ruf02 = ruf02 + 0.1;      // Verboten
```

Man sagt auch: Die Rerferenzvariable (oder: der Alias-Name) `ruf01` bietet eine **unveränderbare** Ansicht (a constant view) der Variablen `vf`, `rvf01` bietet dagegen eine **veränderbare** Ansicht der selben Variablen.

Aufgabe 1: Begründen Sie, warum die Vereinbarung der Referenzvariablen `rvf02` (oben in Zeile 58) verboten ist (und deshalb "auskommentiert" wurde). Was für eine Ansicht (veränderbar oder unveränderbar?) würde die Referenzvariable `rvf02` auf was für eine Variable (veränderbar oder unveränderbar) bieten?

Man beachte, dass es ***keine*** Typen namens `float & const` ("Konstante Referenz auf `float`") und `float const & const` ("Konstante Referenz auf `const float`") gibt. Für den Typ `float const &` wird häufig ("mild irreführend") die Bezeichnung ***konstante Referenz auf float*** anstelle der korrekten Bezeichnung ***Referenz auf konst float*** verwendet.

Natürlich gilt alles, was hier für den Beispieltyp `float` gesagt wurde, auch für andere Typen wie `int`, `double`, `string` etc.

Meistens werden Referenztypen dazu benützt, große Objekte **per Referenz** (statt **per Wert**) an ein Unterprogramm zu übergeben.

Beispiel-07: Große `string`-Objekte per Referenz an ein Unterprogramm übergeben

```
66  void verarbeite(string const & str) {
67     cout << "str.at(0):             " << str.at(0)             << endl;
68     cout << "str.at(str.size()-1): " << str.at(str.size()-1) << endl;
69  // str.at(17) = 'A'; // Verboten!
70  } // verarbeite
71
72  #define MILLIONEN 1000*1000
73
74  string textA(2*MILLIONEN, '?'); // Grosses string-Objekt
75  string textB(3*MILLIONEN, 'X'); // Grosses string-Objekt
76
77  int main() {
78     verarbeite(textA);
79     verarbeite(textB);
80     ...
81  } // main
```

Das Unterprogramm `verarbeite` hat einen Parameter vom Typ ***Referenz auf konstantes string-Objekt***. Die Ausführung eines Aufrufs wie etwa `verarbeite(textA)` beginnt damit, dass der formale Parameter `str` als Referenzvariable erzeugt und mit dem aktuellen Parameter `textA` initialisiert wird, etwa so:

```
82  string const & str = textA;
```

Danach ist `str` ein Alias-Name für `textA`. Die Erzeugung eines solchen Alias-Namens benötigt wenig Zeit (etwa so viel wie die Erzeugung einer Adressvariablen) und erfordert insbesondere ***nicht***, dass zwei Millionen Fragezeichen kopiert werden. Als Bojen dargestellt sehen `textA` und `str` während der Ausführung des Unterprogramms `verarbeite` etwa so aus:

```
83  |textA|-+-<AC00>--["????? ... ?????"]
84          |
85  |str|-- +
```

Innerhalb des Unterprogramms darf der Parameter `str` nicht verändert werden, weil er vom Typ `string const &` ist. Den Typqualifizierer `const` (oben in Zeile 66) sollte man nur weglassen, wenn man dem Unterprogramm die Veränderung seines Parameters ausdrücklich gestatten will.

9 Die Klassenschablone vector und die Klasse string

9.1 vector-Objekte ("C++-Reihungen")

Die C++-Standardbibliothek stellt dem Programmierer unter anderem die Klassenschablone `vector` zur Verfügung (Kopfdatei `<vector>`). Diese ***Schablone*** kann man mit einem beliebigen ***Typ instantiieren*** und erhält dann eine `vector`-Klasse, z. B. `vector<int>` ("vector von `int`") oder `vector<string>` ("vector von `string`") etc. Jedes Objekt der Klasse `vector<int>` hat Ähnlichkeit mit einer Reihung von `int`-Komponenten, ist aber ***sicherer*** und ***flexibler*** (und manchmal ein bisschen langsamer) als eine Reihung. Entsprechend hat jedes Objekt der Klasse `vector<string>` Ähnlichkeit mit einer Reihung von `string`-Komponenten etc.

Im Zusammenhang mit ***Klassenschablonen*** wie `vector` sind ***3 Abstraktionsebenen*** zu unterscheiden:

Ebene	Beispiel
Ebene 1: Die ***Klassenschablone*** namens	`vector`
Ebene 2: Instanzen dieser Schablone werden als ***vector-Klassen*** bezeichnet, z. B.:	`vector<int>`, `vector<string>, ...`

Ebene 3: Instanzen dieser `vector`-Klassen werden als ***vector-Objekte*** bezeichnet, z. B.:	`vector<int> otto;`
	`vector<int> emil; ...`
	
	`vector<string> anna;`
	`vector<string> berta; ...`

Die wichtigsten Eigenschaften von `vector`-Objekten:

1. Sie sind ***typsicher***. Das heisst: In ein Objekt der Klasse `vector<int>` kann man nur `int`-Werte hineintun, in einem Objekt der Klasse `vector<string>` befinden sich nur `string`-Objekte etc.

2. Die ***Länge*** eines `vector`-Objekts (d. h. die Anzahl der Komponenten, die hineinpassen) kann ***dynamisch geändert*** werden ("`vector`-Objekte sind aus Gummi, nicht aus Beton").

3. Wenn man weitere Komponenten immer nur "hinten anhängt" (mit der Operation `push_back`) geht das meistens sogar sehr ***schnell***. Zusätzliche Komponenten an anderen Stellen einfügen ist möglich, dauert aber ***länger***.

4. Auf die ***Komponenten*** eines `vector`-***Objekts*** kann man (ganz ähnlich wie auf die Komponenten einer ***Reihung***) mit Hilfe von ***Indizes*** zugreifen, und zwar wahlweise ***mit Indexprüfung*** (langsamer aber sicherer) oder ***ohne Indexprüfung*** (schneller aber unsicherer).

Vergleich mit Java: In *C++* ist `vector` eine ***Schablone***, mit der man beliebig viele `vector`-Klassen `vector<int>`, `vector<string>` etc. bilden kann. Die Objekte dieser Klassen sind ***typsicher***. In Java 5 ist `Vector` eine ***Klasse***, die einen rohen Typ `Vector` und viele parametrisierte Typen `Vector<Integer>`, `Vector<String>` darstellt (aber keine Typen wie `Vector<int>` mit einem primitiven Typ wie `int` als generischem Parameter). Die parametrisierten Typen sind typsicher, aber der rohe Typ `Vector` ist nicht typsicher. In C++ kann man keine Objekte der Typs `vector` vereinbaren (weil `vector` kein Typ ist, sondern eine Schablone). In Java kann man Objekte des Typs `Vector` vereinbaren (weil `Vector` ein Typ ist), man sollte das aber möglichst selten tun (weil `Vector` ein roher Typ ist, und mit rohen Typen sollte man sich nicht einlassen :-).

Das folgende Beispielprogramm `Vector01` demonstriert die am häufigsten benutzten Befehle für C++-vector-Objekte (***Vereinbarung*** und ***Initialisierung*** von `vector`-Objekten, Zugriff auf Komponenten mit `[]` und mit `at`, Zuweisung an ein `vector`-Objekt mit = und mit `assign`, die Operationen `push_back`, `back` und `front`).

```
1   // Datei Vector01.cpp
2   /* ------------------------------------------------------------------------
3   Die Klassen vector<int>, vector<string> etc. sind Instanzen der Klassen-
4   Schablone vector. Objekte solcher vector-Klassen werden hier vereinbart und
5   es werden typische Operationen darauf angewendet.
6   Ein vector-Objekt aehnelt stark einer Reihung, aber man kann es jederzeit
7   vergroessern ("flexible Reihung").
8   ------------------------------------------------------------------------ */
9   #include <vector>
10  #include <iostream>
11  #include <string>
12  using namespace std;
13  typedef vector<string>::size_type size_type;
14  // ------------------------------------------------------------------------
15  void put(string name, vector<string> vs) {
16     // Gibt name und vs "in lesbarer Form" zur Standardausgabe aus.
17     cout << name << ": ";
18     for (size_type i=0; i<vs.size(); i++) {
19        cout << vs[i] << " ";
20     }
21     cout << endl;
22  } // put
23  // ------------------------------------------------------------------------
24  void put(string name, vector<int> vi) {
25     // Gibt name und vi "in lesbarer Form" zur Standardausgabe aus.
26     cout << name << ": ";
27     for (size_type i=0; i<vi.size(); i++) {
28        cout << vi[i] << " ";
29     }
30     cout << endl;
31  } // put
32  // ------------------------------------------------------------------------
33  int main() {
34     cout << "Vector01: Jetzt geht es los!" << endl;
35
36     // vector-Objekte verschiedener Typen und Groessen vereinbaren:
37     vector<string> vs1;        // Ein leerer string-Vektor
38     vector<string> vs2(2);     // Ein string-Vektor mit 2 Komponenten
39     vector<int>    vi1(5, 0); // int-Vektor mit 5 Komponenten mit dem Wert 0
40     vector<int>    vi2(vi1);  // int-Vektor mit 5 Komponenten mit dem Wert 0
41     vector<int>    vi3 = vi1; // andere Syntax, gleiche Bedeutung
42
43     // vector-Objekte mit Hilfe von Reihungen initialisieren:
44     string rs1[] = {"Hallo!", "Susi!", "Wie geht es?"};
45     int    ri1[] = {17, 5, 24, 33, 2};
46     vector<string> vs3(rs1, rs1 + sizeof(rs1)/sizeof(rs1[0]) );
47     vector<int>    vi4(ri1, ri1 + sizeof(ri1)/sizeof(ri1[0]) );
48
49     // Zugriff auf Komponenten *ohne* Indexpruefung
50     vs2[0] = "Hallo Susi!";
51     vs2[1] = "Wie geht's?";
52
53     // Zugriff auf Komponenten *mit* Indexpruefung (wird vom
54     // Gnu-Cygnus-Compiler Version 2.95.2 leider noch nicht unterstuetzt,
55     // wohl aber vom Borland-Compiler Version 5.5)
56     vs2.at(0) = "Hallo  Susi!";
57     vs2.at(1) = "Wie geht's?";
58
59     // Zuweisung zwischen vector-Objekten:
60     vs1    = vs2;              // vs1 wird eine Kopie von vs2
61
62     vs2[0] = "Hallo Micky!";  // Zuweisung an vorhandene Komponente
```

```
63     vs1.push_back("So lala!");// Eine neue Komponente anhaengen
64
65     // vector-Objekte (mit einer der put-Prozeduren) ausgeben:
66     put("vs1", vs1);
67     put("vs2", vs2);
68     put("vs3", vs3);
69     put("vi4", vi4);
70
71     // Zugriff auf die letzte Komponente von vs2:
72     vs2.back()      = "How are you?";
73     // Zugriff auf Komponenten der ersten Komponente von vs2:
74     vs2.front()[1] = 'e';      // Aus "Hallo" wird "Hello"
75     vs2.front()[6] = 'N';      // Aus "Micky" wird "Nicky"
76     put("vs2", vs2);           // vs2 ausgeben
77
78     // Das vector-Objekt vi1 bearbeiten und ausgeben:
79     put("vi1", vi1);           // vi1 ausgeben
80     vi1[2] = -17;
81     vi1[0] =  2  * vi1[2] + 3;
82     put("vi1", vi1);           // vi1 ausgeben
83     vi1.assign(3, 22);         // 3 Komponenten mit Wert 22 zuweisen
84     put("vi1", vi1);           // vi1 ausgeben
85
86     cout << "Vector01: Das war's erstmal!" << endl;
87  } // main
88  /* ------------------------------------------------------------------------
89  Ausgabe des Programms Vector01:
90
91  Vector01: Jetzt geht es los!
92  vs1: Hallo Susi! Wie geht's? So lala!
93  vs2: Hallo Micky! Wie geht's?
94  vs3: Hallo! Susi! Wie geht es?
95  vi4: 17 5 24 33 2
96  vs2: Hello Nicky! How are you?
97  vi1: 0 0 0 0 0
98  vi1: -31 0 -17 0 0
99  vi1: 22 22 22
100 Vector01: Das war's erstmal!
101 ------------------------------------------------------------------------ */
```

In der C++-Standardbibliothek gibt es außer `vector` noch zahlreiche weitere Behälterschablonen, z. B. `list` und `deque` (sprich deck, double ended queue, zweiköpfige Schlange). Auf die Komponenten eines `vector`-Behählters kann man mit ***Indizes*** zugreifen (siehe oben z. B. die Zeilen 50 und 56), bei `list`- und `deque`-Behältern geht das nicht. Aber mit sogenannten ***Iteratoren*** kann man auf die Komponenten beliebiger Behälter (`vector`-, `list`-, `deque`-, ... Behälter) zugreifen. Solche Iteratoren machen es möglich, Algorithmen (z. B. Sortieralgorithmen) so abstrakt zu programmieren, dass man sie auf beliebige Behälter (`vector`-, `list`-, `deque`-, ... Behälter) anwenden kann.

Im folgenden Beispielprogramm werden verschiedene Arten von Iterator-Objekten vereinbart und benutzt.

```
1   // Datei Vector02.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert, wie man mit Iteratoren auf Vektoren zugreift und Unterpro-
4   gramme aufruft, die Iteratoren als Parameter erwarten (insert und erase).
5   ------------------------------------------------------------------------ */
6   #include <vector>
7   #include <iostream>
8   #include <string>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  //      Abkuerzungen fuer 5 Typ-Bezeichner vereinbaren:
12  //      Typ-Bezeichner:                        Abkuerzung:
13  typedef vector<string>                         stringVektor;
14  typedef stringVektor::iterator                 svIterator;
15  typedef stringVektor::reverse_iterator         svRueckIterator;
16  typedef stringVektor::const_iterator           svKonstIterator;
17  typedef stringVektor::const_reverse_iterator   svKonstRueckIterator;
18  // ------------------------------------------------------------------------
19  void putVorwaerts(string name, stringVektor & sv) {
20     // Gibt die Komponenten von sv in aufsteigender Indexfolge aus.
21     // (sv ist Ref. auf Variable, sollte besser Ref. auf Konstante sein!)
22     cout << name << " (" << sv.size() << " Komp. vorw.  ): ";
23     for (svIterator svi=sv.begin(); svi<sv.end(); svi++) {
24        cout << *svi << " ";
25     }
26     cout << endl;
27  } // putVorwaerts
28  // Die Iteratoren sv.begin() und sv.end() zeigen auf die erste bzw.
29  // hinter die letzte Komponente von sv.
30  // ------------------------------------------------------------------------
31  void putRueckwaerts(string name, stringVektor const & sv) {
32     // Gibt die Komponenten von sv in absteigender Indexfolge aus.
33     // (sv ist Ref. auf Konstante!)
34     cout << name << " (" << sv.size() << " Komp. rueckw.): ";
35     for (svKonstRueckIterator svi=sv.rbegin(); svi<sv.rend(); svi++) {
36        cout << *svi << " ";
37     }
38     cout << endl;
39  } // putRueckwaerts
40  // Die Iteratoren sv.rbegin() und sv.rend() zeigen auf die letzte bzw.
41  // vor die erste Komponente. "svi++" am Ende von Zeile 35 ist korrekt!
42  // ------------------------------------------------------------------------
43  int main() {
44     cout << "Vector02: Jetzt geht es los!" << endl;
45     stringVektor sv1(2);            // Ein Vektor mit 2 Komponenten
46
47     // Mit diversen Operationen Komponenten in den Vektor sv hineintun:
48     sv1[0] = "rot";
49     sv1[1] = "gruen";
50     sv1.push_back("blau");                // hinten anhaengen
51     sv1.insert(sv1.begin(), "schwarz");   // ganz vorn einfuegen
52     sv1.insert(sv1.end()-1, 3, "weiss"); // 3 mal "weiss" einfuegen
53
54     // Den Vektor sv1 ausgeben:
55     putVorwaerts  ("sv1", sv1);
56
57     // Eine unveraenderbare Kopie sv2 von sv1 vereinbaren und ausgeben:
58     stringVektor const sv2(sv1);    // sv2 wird mit sv1 initialisiert
59  // putVorwaerts  ("sv2", sv2);      // Verboten weil sv2 const ist
60     putRueckwaerts("sv2", sv2);
61
62     // Mit diversen Operationen Komponenten aus dem Vektor sv1 entfernen:
```

```
63     sv1.pop_back();                  // Letzte Komponente entfernen
64     sv1.erase(sv1.begin()+2);        // sv1[2] entfernen
65     svIterator von = sv1.begin()+2;  // von zeigt auf sv1[2]
66     svIterator bis = sv1.end();      // bis zeigt hinter letzte Komp.
67     sv1.erase(von, bis);             // Alle Komp. zwischen von und bis entf.
68                                      // (einschliess. von, ausschliess. bis)
69     // Den Vektor sv1 ausgeben:
70     putVorwaerts  ("sv1", sv1);
71
72     // Alle Komponenten aus sv1 entfernen und sv1 ausgeben:
73     sv1.clear();                     // *alle* Komponenten entfernen
74     putRueckwaerts("sv1", sv1);
75
76     cout << "Vector02: Das war's erstmal!" << endl;
77  } // main
78  /* ------------------------------------------------------------------------
79  Fehlermeldung des Compilers, wenn Zeile 59 kein Kommentar ist:
80
81  Vector02.cpp:59: conversion from `const stringVektor' to `stringVektor &'
82                   discards qualifiers
83  Vector02.cpp:19: in passing argument 2 of `putVorwaerts(basic_string<char,
84  string_char_traits<char>,__default_alloc_template<false,0> >, stringVektor &)'
85  ------------------------------------------------------------------------
86  Ausgabe des Programms Vector02:
87
88  Vector02: Jetzt geht es los!
89  sv1 (7 Komp. vorw.  ): schwarz rot gruen weiss weiss weiss blau
90  sv2 (7 Komp. rueckw.): blau weiss weiss weiss gruen rot schwarz
91  sv1 (2 Komp. vorw.  ): schwarz rot
92  sv1 (0 Komp. rueckw.):
93  Vector02: Das war's erstmal!
94  --------------------------------------------------------------------- */
```

Das folgende Beispielprogramm `Algorithmen01` zeigt die Anwendung von *Iteratoren* als "Bindeglied zwischen Behälter-Schablonen und abstrakten Algorithmen" (für Fortgeschrittene).

```
1   // Datei Algorithmen01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert einige Algorithmen aus der STL (standard template library)
4   fuer Behaelter (am Beispiel von Vectoren). Die Algorithmen sind als Unter-
5   programme realisiert (for_each, transform, sort, remove_if etc.), die
6   Iteratoren und Unterprogramme als Parameter erwarten.
7   --------------------------------------------------------------------- */
8   #include <iostream>
9   #include <vector>
10  #include <string>
11  #include <algorithm> // fuer for_each, transform, sort, remove_if
12  #include <iomanip>   // fuer setw
13  #include <cstdlib>   // fuer rand
14  using namespace std;
15  // ---------------------------------------------------------------------
16  //      Abkuerzungen fuer drei Typ-Bezeichner vereinbaren:
17  //      Typ-Bezeichner:              Abkuerzung:
18  typedef vector<int>                  intVektor;
19  typedef intVektor::iterator          ivIterator;
20  typedef intVektor::const_iterator    ivKonstIterator;
21  // ---------------------------------------------------------------------
22  void put(string kommentar, intVektor const & iv) {
23     // Gibt den kommentar und iv "in lesbarer Form" zur Standardausgabe aus.
24     cout << kommentar << ": ";
25     for (ivKonstIterator ivi=iv.begin(); ivi<iv.end(); ivi++) {
26        cout << setw(4) << *ivi;
27     }
```

```
28      cout << endl;
29   } // put
30   // ------------------------------------------------------------------------
31   void fuelleZufaellig(int & n) {   // Prozedur mit Ref. auf int Parameter
32      n = rand() % 1000;
33   } // fuelleZufaellig
34   // ------------------------------------------------------------------------
35   int mod100(int n) {               // int-Funktion mit int-Parameter
36      return n % 100;
37   } // mod100
38   // ------------------------------------------------------------------------
39   bool istGerade(int n) {           // bool-Funktion mit int Parameter
40      return n % 2 == 0;
41   } // istGerade
42   // ------------------------------------------------------------------------
43   bool sindAbsteigendSortiert(int n1, int n2) { // Vergleichsfunktion
44      return n2 < n1;
45   } // sindAbsteigendSortiert
46   // ------------------------------------------------------------------------
47   bool sindSonderbarSortiert(int n1, int n2) {  // Vergleichsfunktion
48      if (n1 % 2  > n2 % 2) return false;
49      if (n1 % 2  < n2 % 2) return true;
50      if (n1 % 2 == 1)      return n2 < n1;
51      if (n1 % 2 == 0)      return n1 < n2;
52      return true; // Nur damit der Borland-Compiler nicht nervoes wird!
53   } // sindSonderbarSortiert
54   // ------------------------------------------------------------------------
55   int main() {
56      cout << "Algorithmen01: Jetzt geht es los!" << endl;
57      intVektor v1(10);
58
59      srand(576); // Zufallszahlengenerator initialisieren
60
61      // v1 mit Zufallszahlen fuellen und ausgeben:
62      for_each(v1.begin(), v1.end(), fuelleZufaellig);
63      put("v1 (Zufallszahlen)    ", v1);
64
65      // Jede Komponente k von v1 durch mod100(k) ersetzen:
66      transform(v1.begin(), v1.end(), v1.begin(), mod100);
67      put("v1 (mod100)           ", v1);
68
69      // v1 aufsteigend sortieren:
70      sort(v1.begin(), v1.end());
71      put("v1 (aufsteigend sort.)", v1);
72
73      // v1 absteigend sortieren:
74      sort(v1.begin(), v1.end(), sindAbsteigendSortiert);
75      put("v1 (absteigend  sort.)", v1);
76
77      // v1 "sonderbar" sortieren (erst alle geraden Zahlen aufsteigend,
78      // dann alle ungeraden Zahlen absteigend):
79      sort(v1.begin(), v1.end(), sindSonderbarSortiert);
80      put("v1 (sonderbar   sort.)", v1);
81
82      // Aus v2 (einer Kopie von v1) alle geraden Zahlen entfernen:
83      intVektor v2(v1);
84      ivIterator bis = remove_if(v2.begin(), v2.end(), istGerade);
85      put("v2 (nach remove_if()) ", v2);
86      v2.erase(bis, v2.end());
87      put("v2 (nach erase())     ", v2);
88
89      cout << "Algorithmen01: Das war's erstmal!" << endl;
```

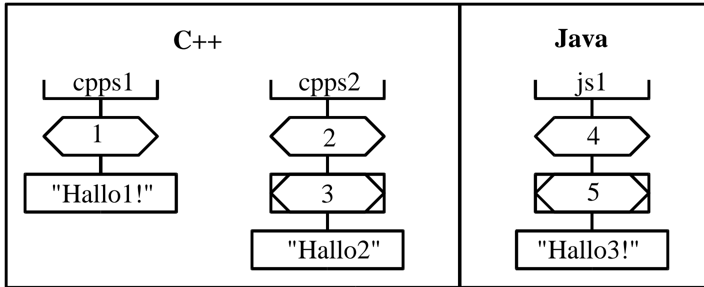
```
90  } // main
91  /* ------------------------------------------------------------------------
92  Ausgabe des Programms Algorithmen01:
93
94  Algorithmen01: Jetzt geht es los!
95  v1 (Zufallszahlen)    :  324 184 256 991 740 507 204 439 538 172
96  v1 (mod100)           :   24  84  56  91  40   7   4  39  38  72
97  v1 (aufsteigend sort.):    4   7  24  38  39  40  56  72  84  91
98  v1 (absteigend  sort.):   91  84  72  56  40  39  38  24   7   4
99  v1 (sonderbar   sort.):    4  24  38  40  56  72  84  91  39   7
100 v2 (nach remove_if()) :   91  39   7  40  56  72  84  91  39   7
101 v2 (nach erase())     :   91  39   7
102 Algorithmen01: Das war's erstmal!
103 ------------------------------------------------------------------------ */
```

Die Programme `Algorithmen02` und `Algorithmen03` (hier nicht wiedergegeben) enthalten weitere Beispiele zum Thema ***abstrakte Algorithmen auf Objekte beliebiger Behälterklassen anwenden***. Mit Hilfe der Behälter-Schablone `deque` (double ended queue) kann man Behälterobjekte erzeugen, bei denen das ***Einfügen zusätzlicher Komponenten*** sowohl ***vorn*** als auch ***hinten*** relativ ***schnell*** geht. Das Programm `DeQue01` (hier nicht wiedergegeben) demonstriert einige Möglichkeiten der Klassen-Schablone `deque`.

9.2 String-Objekte ("C++-Strings")

Zwischen der ***Java***-Klasse `String` und der ***C++***-Klasse `string` gibt es viele ***Ähnlichkeiten*** und einen wichtigen ***Unterschied***. Die folgenden Vereinbarungen und Bojen sollen den ***Unterschied*** deutlich machen:

```
1    // C++-Vereinbarungen:                      // Java-Vereinbarung:
2    string   cpps1("Hallo1!");                  String js1 = new String("Hallo3");
3    string * cpps2 = new string("Hallo2!");
```



Das folgende Beispielprogramm `CppStrings01` soll die wichtigsten Eigenschaften von ***C++-Strings*** (und ihre ***Ähnlichkeit*** zu ***Java-Strings***) deutlich machen:

```
1   // Datei CppStrings01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert den Umgang mit C++-Strings: Strings vereinbaren und initiali-
4   sieren, Werte zuweisen, auf Komponenten zugreifen, Strings und Zeichen kon-
5   katenieren, Zeichen einfuegen (insert), Zeichen entfernen (erase), Zeichen
6   ersetzen (replace), Teilstrings bilden (substr), Strings vergleichen (mit <,
7   <=, == etc. und mit compare) und einen C++-String in einen C-String umwan-
8   deln (c_str). *Nicht* demonstriert wird das Suchen in einem String (find).
9   ------------------------------------------------------------------------ */
```

```
10 #include <string>     // size, length, capacity, [], at, clear,
11 #include <iostream>   // cout, cin, <<, >>, endl
12 #include <iomanip>    // setw, setprecision, setiosflags, setbase, <ios>
13 using namespace std;
14
15 int main() {
16    // C++-String-Variablen vereinbaren und initialisieren (mit diversen
17    // Konstruktoren). Die Variablen a01 bis a06 haben gleiche Anfangswerte:
18    string a01 = "ABCDEFGH";         // Initialis. mit Literal ist erlaubt
19    string a02 =  a01;               // Initialis. mit C++-String auch.
20    string a03("ABCDEFGH");          // Alternative (Konstruktor-) Notation
21    string a04(a03);                 // Alternative (Konstruktor-) Notation
22    string a05(a04, 0, 8);           // Von a04 acht       Zeichen ab Index 0
23    string a06(a05, 0, a05.size()); // Von a05 a05.size() Zeichen ab Index 0
24    string a07;                      // Ein leerer String (null Zeichen)
25    string a08(50, 'X');             // 50 mal das Zeichen 'X'
26 // string a09 = 'X';                // Initialis. mit char ist nicht erlaubt
27 // string a10('X');                 // Initialis. mit char ist nicht erlaubt
28
29    // Leere Ergebnis-Strings vereinbaren:
30    string e01, e02, e03, e04, e05, e06, e07, e08, e09, e10, e11, e12, e13;
31    string e14, e15, e16, e17, e18, e19, e20, e21, e22, e23, e24, e25, e26;
32
33    char c01='?', c02=c01, c03=c01, c04=c01, c05=c01, c06=c01, c07='?';
34
35    // Einer string-Variablen einen Wert zuweisen:
36    e01 = "ABCDEFGH";                // Zuweisung mit Literal ist erlaubt
37    e02 =  e01;                      // Zuweis. mit C++-String natuerlich auch
38    e03 = 'X';                       // Zuweis. mit char ist erlaubt!!!
39
40    e04.assign("ABCDEFGH");          // Zuweisung mit Literal ist erlaubt
41    e05.assign(e01);                 // Zuweis. mit C++-String natuerlich auch
42    e06.assign(e01, 2, 3);           // Von e01 drei Zeichen ab Index 2
43    e07.assign(50, '-');             // 50 mal das Zeichen '-'
44
45    // Auf Komponenten eines Strings zugreifen (*ohne* Indexpruefung):
46    e08 = "abcdefgh";
47    c01 = e08[0];                    // Erstes  Zeichen von e08 ('a')
48    c02 = e08[e08.size()-1];         // Letztes Zeichen von e08 ('h')
49    c03 = e08[50];                   // Ohne Indexpruefung!!!
50    e08[0]             = 'A';        // Erstes  Zeichen ersetzen
51    e08[e08.size()-1] = 'H';         // Letztes Zeichen ersetzen
52    e08[50]            = c03;        // Ohne Indexpruefung!!!
53
54    // Auf Komponenten eines Strings zugreifen ( *mit* Indexpruefung):
55    e09 = "abcdefgh";
56    c04 = e09.at(0);                 // Erstes  Zeichen von e09 ('a')
57    c05 = e09.at(e09.size()-1);      // Letztes Zeichen von e09 ('h')
58 // c06 = e09.at(50);                // Loest eine Ausnahme aus
59    e09.at(0)              = 'A';    // Erstes  Zeichen ersetzen
60    e09.at(e09.size()-1) = 'H';      // Letztes Zeichen ersetzen
61 // e09.at(50)             = c06;    // Loest eine Ausnahme aus
62
63    // Strings und einzelne Zeichen (vom Typ char) konkatenieren:
64    e10  = "Hallo! ";
65    e10  = e10 + " Wie geht's" + c07 + '!' + c07; // "Hallo! Wie geht's ?!?"
66    e11  = "Na ";
67    e11 += "ja.";                    // Mit String "ja.", Ergeb.: "Na ja."
68    e12  = "So la la";
69    e12 += '!';                      // Mit char '!'    , Ergeb.: "So la la!"
70    e13  = "Hallo Su";
71    e13.append("si!");               // "Hallo Susi! "
```

```
72      e14  = "Hallo Susi! ";
73      e15  = "geht's? Wie ";
74      e14.append(e15, 8, 4).append(e15, 0, 7); // "Hallo Susi! Wie geht's?"
75
76      // Zeichen in einen String einfuegen (mit insert):
77      e15 = "Hallo, wie geht es?";
78      e15.insert(5, " Susi");         // Alle Zeichen  von " Susi"
79      e16 = "Hello, how are you?";
80      e16.insert(5, " Susi", 3);      // Nur 3 Zeichen von " Susi"
81
82      // Zeichen aus einem String entfernen (mit erase):
83      e17 = "ABC123abc";
84      e18 = "ABC123abc";
85      e19 = "ABC123abc";
86      e17.erase(2, 5);                // Ab Index 2 werden 5 Zeichen entfernt
87      e18.erase(6);                   // Ab Index 6 werden alle Zeichen entf.
88      e19.erase();                    // Alle Zeichen werden entfernt
89
90      // Zeichen eines Strings ersetzen (mit replace):
91      e20 = "Hallo Susi!";
92      e20.replace(6, 4, "Susanna");   // Alle Zeichen  von "Susanna"
93      e21 = "Hallo Susi!";
94      e21.replace(6, 4, "Susanna", 2);// Nur 2 Zeichen von "Susanna"
95
96      // Einen Teilstring eines Strings bilden:
97      e23 = "Hallo Axel!";
98      e22 = e23.substr(6, 4);         // "Axel"
99
100     // Strings vergleichen (siehe unten die Ausgabebefehle):
101     e23 = "abc";                    //             "abc"   ist
102     e24 = "abcde";                  // kleiner als "abcde" und
103     e25 = "abc";                    // gleich      "abc"
104
105     // Einen ("modernen") C++-String in einen ("alten") C-String umwandeln:
106     e26              = "Hallo!";    // e26 ist ein C++-String
107     const char * e27 = e26.c_str(); // e27 ist ein C-String
108     // ------------------------------------------------------------------------
109     // Die Ergebnisse ausgeben:
110     cout << boolalpha;
111     cout << "A e01: " << e01 << ", e02: " << e02 << ", e03: " << e03 << endl;
112     cout << "B e04: " << e04 << ", e05: " << e05 << ", e06: " << e06 << endl;
113     cout << "C e07: " << e07                                         << endl;
114     cout << "D c01: " << c01 << ", c02: " << c02 << ", c03: " << c03 << endl;
115     cout << "E e08: " << e08                                         << endl;
116     cout << "F c04: " << c04 << ", c05: " << c05 << ", c06: " << c06 << endl;
117     cout << "G e09: " << e09                                         << endl;
118     cout << "H e10: " << e10 << ", e11: " << e11 << ", e12: " << e12 << endl;
119     cout << "I e13: " << e13 << ", e14: " << e14                     << endl;
120     cout << "J e15: " << e15 << ", e16: " << e16                     << endl;
121     cout << "K e17: " << e17 << ", e18: " << e18 << ", e19: " << e19 << endl;
122     cout << "L e20: " << e20 << ", e21: " << e21                     << endl;
123     cout << "M e22: " << e22                                         << endl;
124     cout << "N e23: " << e23 << ", e24: " << e24 << ", e25: " << e25 << endl;
125     cout << "O (e23 == e22)    : " << (e23 == e22)                   << endl;
126     cout << "P (e23 >= e21)    : " << (e23 >= e21)                   << endl;
127     cout << "Q (e23 <  e21)    : " << (e23 <  e21)                   << endl;
128     cout << "R e23.compare(e21): " << e23.compare(e21)               << endl;
129     cout << "S e21.compare(e23): " << e21.compare(e23)               << endl;
130     cout << "T e23.compare(e22): " << e23.compare(e22)               << endl;
131     cout << "U e26: " << e26 << ", e27: " << e27                     << endl;
132     // ------------------------------------------------------------------------
133 } // main
134 /* ------------------------------------------------------------------------
```

```
135 Ausgabe des Programms CppStrings01:
136
137 A e01: ABCDEFGH, e02: ABCDEFGH, e03: X
138 B e04: ABCDEFGH, e05: ABCDEFGH, e06: CDE
139 C e07: ----------------------------------------------------
140 D c01: a, c02: h, c03: ]
141 E e08: AbcdefgH
142 F c04: a, c05: h, c06: ?
143 G e09: AbcdefgH
144 H e10: Hallo!  Wie geht's?!?, e11: Na ja., e12: So la la!
145 I e13: Hallo Susi!, e14: Hallo Susi! Wie geht's?
146 J e15: Hallo Susi, wie geht es?, e16: Hello Su, how are you?
147 K e17: ABbc, e18: ABC123, e19:
148 L e20: Hallo Susanna!, e21: Hallo Su!
149 M e22: Axel
150 N e23: abc, e24: abcde, e25: abc
151 O (e23 == e22)    : false
152 P (e23 >= e21)    : true
153 Q (e23 <  e21)    : false
154 R e23.compare(e21): 1
155 S e21.compare(e23): -1
156 T e23.compare(e22): 1
157 U e26: Hallo!, e27: Hallo!
158 ------------------------------------------------------------------------------ */
```

10 Generische Einheiten, Teil 1 (Unterprogrammschablonen, function templates)

10.1 Einleitung und Motivation

Die Grundidee eines ***starken Typensystems***: Jeder Ausdruck und sein Wert gehören zu einem bestimmten ***Typ***. Auf Ausdrücke und Werte eines bestimmten Typs darf man nur bestimmte ***Operationen*** anwenden. Z. B. darf man ***string***-Werte ***konkatenieren*** (aber nicht multiplizieren), ***int***-Werte darf man ***multiplizieren*** (aber nicht konkatenieren), ein Unterprogramm mit einem formalen ***string***-Parameter darf man nur mit einem aktuellen ***string***-Parameter aufrufen etc. Mit solch einem Typensystem werden viele ***Flüchtigkeitsfehler*** des Programmierers zu ***Typfehlern***, die der Ausführer schon bei der Übergabe des Programms ("zur Compilezeit") ***automatisch entdecken*** kann.

Ein starkes Typensystem ***verbilligt*** das Finden vieler Programmierfehler und macht Programme ***sicherer***. Leider ist dieser Schutz vor Flüchtigkeitsfehlern aber manchmal auch ***lästig*** und unangenehm einschränkend. Viele abstrakte ***Algorithmen*** funktionieren gleichermaßen für ***Werte verschiedener Typen***. Das lässt sich aber in einer stark getypten Sprache nicht ausdrücken (zumindest nicht ohne weiteres).

Betrachten wir als Beispiel eine simple Funktion zum Potenzieren von Ganzzahlen:

```
1   int hoch(int basis, int exponent) {
2      int erg = 1;
3      for (int i=1; i<=exponent; i++) erg *= basis;
4      return erg;
5   }
```

Eine Prüfung der Parameter (exponent negativ? etc.) und andere durchaus sinnvolle Verfeinerungen wurden hier nicht programmiert, um das Beispiel möglichst einfach zu lassen.

Diese Funktion `hoch` ist auf den Typ `int` festgelegt. Das ist schade, weil der abstrakte Algorithmus, den sie realisiert, genauso gut auch für `long`-Werte oder `short`-Werte etc. funktionieren würde. Den Text der Funktion zu ***kopieren*** und alle Vorkommen von `int` mit einem Editor durch `long` oder durch `short` etc. zu ersetzen, ist zwar möglich, aber aus verschiedenen Gründen ***keine*** empfehlenswerte Vorgehensweise (beim Manipulieren der Texte können sich ***Fehler*** einschleichen, man muss mehrere Kopien der Funktion manuell verwalten etc.).

10.2 Unterprogrammschablonen (function templates)

Mit ***generischen Einheiten*** (in C++: mit ***Schablonen***, engl. ***templates***) kann man unter anderem Algorithmen beschreiben und dabei ***abstrakte Typparameter*** anstelle von ***konkreten Typen*** verwenden. Statt einer genaueren Definition hier ein konkretes Beispiel für eine generische Einheit:

```
1   template<typename Ganz>
2   Ganz hoch(Ganz basis, Ganz exponent) {
3      Ganz erg = 1;
4      for (Ganz i=1; i<=exponent; i++) erg *= basis;
5      return erg;
6   }
```

Dies ist die Definition einer Funktionsschablone namens `hoch` mit einem ***Typparameter*** namens `Ganz`. Namen wie `hoch<int>`, `hoch<long>` etc. bezeichnen ***Instanzen*** dieser Schablone. Jede solche Instanz ist eine ***Funktion*** (weil `hoch` eine ***Funktions***schablone ist). Die ***Definition*** der Funktion `hoch<int>` erzeugt der Ausführer automatisch, indem er in der Schablone `hoch` (genauer: in einer Kopie der Zeilen 2 bis 6) den formalen Schablonenparameter `Ganz` durch den konkreten Typ `int` ersetzt. Für die Funktionen `hoch<long>`, `hoch<short>` etc. gilt entsprechendes. ***Aufrufen*** kann man diese Funktionen z. B. so:

```
7   int main() {
8      int  g1 = hoch<int>(2, 3);     // hoch wird mit dem Typ int  instanziiert
9      long g2 = hoch<long>(2L, 3L); // hoch wird mit dem Typ long instanziiert
10     long g3 = hoch      (g1, 2);  // hoch wird mit dem Typ int instanziiert
11     long g4 = hoch      (g1, g2); // Fehler, Typ nicht eindeutig erkennbar
```

Beim Instanziieren der Schablone `hoch` darf man auf die ausdrückliche Angabe eines konkreten Typs (in spitzen Klammern, z. B. `<int>`) ***verzichten***, wenn der Ausführer aus dem Zusammenhang erkennen kann, welcher konkrete Typ gemeint ist (siehe Zeile 13).

Wenn man Schablonen verwendet, dann übernimmt der ***Ausführer*** das Kopieren der Schablonen, das Einsetzen der aktuellen Parameter und das Verwalten der verschiedenen Kopien (Instanzen) der Schablone. ***Maschinelle Ausführer*** machen in aller Regel deutlich weniger Flüchtigkeitsfehler als ***menschliche Programmierer***.

10.3 Der Unterschied zwischen Parametern und Parametern

Im Zusammenhang mit Unterprogrammschablonen spielen ***zwei Arten von Parametern*** eine Rolle: ***Schablonenparameter*** und ***Unterprogrammparameter***. Diese Tatsache macht eine Reihe von Begriffsverwirrungen und Verständnisproblemen möglich. Hier ein Versuch, diese Verwirrungen und Probleme zu vermeiden, zu beseitigen oder zumindest zu verkleinern:

1. Eine ***Schablone*** hat (null oder mehr) ***Schablonenparameter***. Z. B. hat die Schablone `hoch` genau ***einen*** Schablonenparameter namens `Ganz`.

2. Ein ***Unterprogramm*** hat (Unterprogramm-) ***Parameter***. Z. B. hat das Unterprogramm `hoch<int>` genau ***zwei*** (Unterprogramm-) Parameter namens `basis` und `exponent`. Das Gleiche gilt auch für die Unterprogramme `hoch<long>`, `hoch<short>` etc.

3. Alle mit ***Schablonen*** zusammenhängenden Arbeiten (das ***Übergeben*** von ***Schablonenparameter*** an eine Schablone und das ***Instanziieren*** einer ***Schablone***) werden vom Ausführer schon bei der ***Übergabe des Programms*** ("zur Compilezeit") durchgeführt (und kosten somit keine Ausführungszeit).

4. ***Unterprogramme*** werden erst bei der ***Ausführung*** eines Programms ("zur Laufzeit") ***aufgerufen***, mit ***Parametern versehen*** und ***ausgeführt***.

Das folgende Diagramm ist ein Versuch, den Zusammenhang zwischen den Begriffen ***Unterprogramm-Schablone***, ***Unterprogramm***, ***Schablonenparameter***, ***Unterprogrammparameter***, ***instanziieren*** und ***aufrufen*** zu verdeutlichen:

	Beispiel(e):		
Eine **Unterprogramm-Schablone** hat **Schablonen**-Parameter	`hoch` `Ganz`		
↓ wird **instanziiert** (zur **Compilezeit**)			
Jede Instanz ist ein **Unterprogramm** und hat **Unterprogramm**-Parameter	`hoch<int>` `basis, exponent`	`hoch<long>` `basis, exponent`	... ...
↓ wird **aufgerufen** (zur **Laufzeit**)	`hoch<int>(2, 3)`	`hoch<long>(2L, 10L)`	
Ergebnis der **Funktion** bzw. **Seiteneffekt** der **Prozedur**	8	1024	

Klassenschablonen werden später behandelt (siehe Abschnitt 13).

10.4 Neue Möglichkeiten, Programmierfehler zu machen

Der Programmierer der Schablone `hoch` wollte eigentlich, dass für den Schablonenparameter `Ganz` nur ***Ganzzahltypen*** (`short`, `int`, `long`, `unsigned`, ...) eingesetzt werden. Er hat das aber nicht zwingend vorgeschrieben, so dass man beim Instanziieren der Schablone auch ***ungeeignete*** Typen wie `float`, `bool`, `string` etc. angeben kann. Der Compiler meldet dabb in einigen Fällen einen Fehler, aber in anderen Fällen erzeugt er (ohne Warnung) eine ***unsinnige Instanz*** der Schablone `hoch`. Siehe dazu im folgenden Abschnitt das Beispielprogramm `Schablonen01` und die darauffolgende Aufgabe sowie weiter unten den Abschnitt über Metaprogrammierung.

10.5 Das Beispielprogramm Schablonen01 (die Schablone hoch)

```
1   // Datei Schablonen01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert eine Funktionsschablone namens hoch  mit deren Instanzen man
4   Werte beliebiger Ganzzahltypen (short, int, long, ...) potenzieren kann.
5   ------------------------------------------------------------------------ */
6   #include <iostream>
7   #include <string>
8   using namespace std;
9   // ------------------------------------------------------------------------
10  // Eine Funktionsschablone zum Potenzieren von Ganzzahlen
11  // (vereinfachte Version ohne Fehlerpruefungen):
12  template <typename Ganz>
13  Ganz hoch(Ganz basis, Ganz exponent) {
14     Ganz erg = 1;
15     for (Ganz i=1; i<=exponent; i++) erg *= basis;
16     return erg;
17  } // template hoch
18  // ------------------------------------------------------------------------
19  int main() {
20     cout << "Schablonen01: Jetzt geht es los!" << endl;
21
22     int    const I1 = 2;
23     int    const I2 = 3;
24     long   const L1 = 3;
25     float  const F1 = 3;
26     float  const F2 = 2.2;
27     bool   const B1 = false;
28     string const S1 = "2";
29
30     cout << "hoch      (I1, I2) : " << hoch      (I1, I2) << endl;
31     cout << "hoch      (L1, L1) : " << hoch      (L1, L1) << endl;
32     cout << "hoch      (I2,  2) : " << hoch      (I2,  2) << endl;
33  // cout << "hoch      (L1, I1) : " << hoch      (L1, I1) << endl;
34  // cout << "hoch      (L1,  2) : " << hoch      (L1,  2) << endl;
35     cout << "hoch      (L1, 2L) : " << hoch      (L1, 2L) << endl;
36     cout << "hoch<long>(L1,  2) : " << hoch<long>(L1,  2) << endl;
37     cout << "hoch      (F1, F1) : " << hoch      (F1, F1) << endl;
38     cout << "hoch      (F2, F2) : " << hoch      (F2, F2) << endl;
39     cout << "hoch      (B1, B1) : " << hoch      (B1, B1) << endl;
40  // cout << "hoch      (S1, S1) : " << hoch      (S1, S1) << endl;
41
42     cout << "Schablonen01: Das war's erstmal!" << endl;
43  } // main
44  /* ------------------------------------------------------------------------
```

```
45  Fehlermeldungen des Gnu-Cygnus-Compilers (Version 2.95.2), wenn die Zeilen
46  33 und 34 keine Kommentare sind:
47
48  Schablonen01.cpp:33: no matching function for call to `hoch (long &, int &)'
49  Schablonen01.cpp:34: no matching function for call to `hoch (long &, int)'
50  ------------------------------------------------------------------------
51  Fehlermeldung des Compilers, wenn Zeile 40 kein Kommentar ist (gekuerzt):
52
53  Schablonen01.cpp: In function
54     `class basic_string<char,string_char_traits<char>,
55     __default_alloc_template<false,0> > hoch<basic_string
56     <char,string_char_traits<char>,__default_alloc_template<false,0> > >
57     (basic_string<char,string_char_traits<char>,__default_alloc_template
58     <false,0> >, basic_string<char,string_char_traits
59     <char>,__default_alloc_template<false,0> >)':
60  Schablonen01.cpp:40:   instantiated from here
61  Schablonen01.cpp:14: conversion from `int' to non-scalar type ... u.s.w.
62  ------------------------------------------------------------------------
63  Ausgabe des Programms Schablonen01:
64
65  Schablonen01: Jetzt geht es los!
66  hoch      (I1, I2) : 8
67  hoch      (L1, L1) : 27
68  hoch      (I2,  2) : 9
69  hoch      (L1, 2L) : 9
70  hoch<long>(L1,  2) : 9
71  hoch      (F1, F1) : 27
72  hoch      (F2, F2) : 4.84
73  hoch      (B1, B1) : 1
74  Schablonen01: Das war's erstmal!
75  --------------------------------------------------------------------- */
```

Aufgabe: Auf meinem Taschenrechner ist $2.2^{2.2}$ gleich `5.666695779`, dagegen ist (oben, Zeile 72) `hoch(F2, F2)` gleich `4.84`! Sollte ich mir einen neuen Taschenrechner kaufen?

10.6 Das Beispielprogramm Schablonen02 (mehrere Schablonen put)

Schablonen können nicht nur ***Typ***parameter haben, sondern auch "normale" Parameter (***Werte*** beliebiger Typen). Außerdem zeigt das folgende Beispielprogramm, dass ***ein*** Name ***mehrere*** Schablonen und Unterprogramme bezeichnen kann (Namen von Unterprogrammen und Schablonen können ***überladen*** werden). Die betreffenden Schablonen und das Unterprogramm namens `put` werden im Kommentar mit den zusätzlichen (und eindeutigen) Namen `P1`, `S1`, `S2` und `S3` auseinandergehalten.

```
1   // Datei Schablonen02.cpp
2   /* ------------------------------------------------------------------------
3   Ein formaler Unterprogramm-Parameter r kann zu einem Typ "Referenz auf
4   eine Reihung" gehoeren. Ein solcher Parameter muss aber nicht nur den
5   Komponententyp der Reihung, sondern auch ihre Laenge festlegen:
6   Erlaubt:  void put(int (&r)[5]); // Referenz auf Reihung der Laenge 5
7   Verboten: void put(int (&r)[ ]); // Laengenangabe fehlt
8   Statt fuer jede moegliche Laenge ein eigenes Unterprogramm zu schreiben,
9   kann man eine Schablone mit einem Laengen-Parameter vereinbaren. Noch
10  nuetzlicher ist eine Schablone, in der auch der Komponententyp des
11  (Referenz-auf-eine-) Reihung-Parameters ein (Schablonen-) Parameter ist.
12  --------------------------------------------------------------------- */
13  #include <iostream>
14  using namespace std;
15  //------------------------------------------------------------------------
16  // Eine Prozedur und drei Prozedurschablonen namens put werden deklariert:
17  // -----------------------------------------------------------------------
18  void put(int (& r)[5]);     // Prozedur P1:  fuer int-Reihungen mit Laenge 5
19
20  template <class T>          // Schablone S1 mit Typparameter T
21  void put(T (& r)[5]);       // fuer T-Reihungen mit Laenge 5
22
23  template <int L>            // Schablone S2 mit int-Parameter L
24  void put(int (& r)[L]);     // fuer int-Reihung mit Laenge L
25
26  template <class T, int L>   // Schablone S3 mit Typparam. T und int-Param L
27  void put(T (& r)[L]);       // fuer T-Reihungen mit Laenge L
28  //------------------------------------------------------------------------
29  int main() {
30     cout << "Schablonen02: Jetzt geht es los!" << endl;
31     int   ir5[5] = {51, 52, 53, 54, 55};
32     int   ir7[7] = {71, 72, 73, 74, 75, 76, 77};
33     char  cr5[5] = {'A', 'B', 'C', 'D', 'E'};
34     char  cr6[6] = {'a', 'b', 'c', 'd', 'e', 'f'};
35     bool  br3[3] = {true, false, true};
36     float fr4[4] = {1.5, 2.5, 3.5, 4.5};
37     put(ir5); // put ist Prozedur               P1
38     put(cr5); // S1 oder S3? Gnu nimmt S1, BCC meldet Fehler.
39     put(ir7); // put ist Instanz von Schablone S2 (mit             L := 7)
40     put(cr6); // put ist Instanz von Schablone S3 (mit T := char,  L := 6)
41     put(br3); // put ist Instanz von Schablone S3 (mit T := bool,  L := 3)
42     put(fr4); // put ist Instanz von Schablone S3 (mit T := float, L := 4)
43     cout << "Schablonen02: Das war's erstmal!" << endl;
44  } // main
45  //------------------------------------------------------------------------
46  // Die Prozedur und die drei Prozedurschablonen namens put werden definiert:
47  // -----------------------------------------------------------------------
48  void put(int (& r)[5]) { // Komponententyp int, Laenge 5
49     // Gibt die Reihung r in lesbarer Form zur Standardausgabe aus:
50     cout << "Proz.  P1: Eine Reihung mit 5 Komponenten: ";
51     for (int i=0; i < 5; i++) {
52        cout << r[i] << " ";
53     }
```

```
54      cout << endl;
55   } // put (alias P1)
56   //------------------------------------------------------------------------
57   template <class T>
58   void put(T (& r)[5]) { // beliebiger Komponententyp T, Laenge 5
59      // Gibt die Reihung r in lesbarer Form zur Standardausgabe aus:
60      cout << "Schab. S1: Eine Reihung mit 5 Komponenten: ";
61      for (int i=0; i < 5; i++) {
62         cout << r[i] << " ";
63      }
64      cout << endl;
65   } // put (alias S1)
66   //------------------------------------------------------------------------
67   template <int L>
68   void put(int (& r)[L]) { // Komponententyp int, beliebige Laenge L
69      // Gibt die Reihung r in lesbarer Form zur Standardausgabe aus:
70      cout << "Schab. S2: Eine Reihung mit " << L << " Komponenten: ";
71      for (int i=0; i < L; i++) {
72         cout << r[i] << " ";
73      }
74      cout << endl;
75   } // put (alias S2)
76   //------------------------------------------------------------------------
77   template <class T, int L>
78   void put(T (& r)[L]) { // beliebiger Komponententyp T, beliebige Laenge L
79      // Gibt die Reihung r in lesbarer Form zur Standardausgabe aus:
80      cout << "Schab. S3: Eine Reihung mit " << L << " Komponenten: ";
81      for (int i=0; i < L; i++) {
82         cout << r[i] << " ";
83      }
84      cout << endl;
85   } // put (alias S3)
86   /* ------------------------------------------------------------------------
87   Ausgabe des Programms Schablonen02:
88
89   Schablonen02: Jetzt geht es los!
90   Proz.  P1: Eine Reihung mit 5 Komponenten: 51 52 53 54 55
91   Schab. S1: Eine Reihung mit 5 Komponenten: A B C D E
92   Schab. S2: Eine Reihung mit 7 Komponenten: 71 72 73 74 75 76 77
93   Schab. S3: Eine Reihung mit 6 Komponenten: a b c d e f
94   Schab. S3: Eine Reihung mit 3 Komponenten: 1 0 1
95   Schab. S3: Eine Reihung mit 4 Komponenten: 1.5 2.5 3.5 4.5
96   Schablonen02: Das war's erstmal!
97   ------------------------------------------------------------------------
98   Falls Zeile 18 zu einem Kommentar gemacht wird:
99
100  Der Gnu-Cynus-Compiler, Version 2.95.2, meldet folgenden Fehler:
101
102  Schablonen02.cpp: In function `int main()':
103  Schablonen02.cpp:37: call of overloaded `put (int[5])' is ambiguous
104  Schablonen02.cpp:21: candidates are: void put<int>(int (&)[5])
105  Schablonen02.cpp:24:                 void put<5>(int (&)[5])
106  Schablonen02.cpp:27:                 void put<int, 5>(int (&)[5])
107
108  Der Borland-Compiler, Version 5.5. meldet keinen Fehler, sondern nimmt
109  in Zeile 37 (fuer put(ir5);) die Schablone S2.
110  ---------------------------------------------------------------------- */
```

Namensauflösung (name resolution): Ein Unterprogramm-Name kann ***überladen*** sein, d. h. gleichzeitig ***mehrere*** verschiedene Unterprogramme bezeichnen. Ein Unterprogramm-Name kann in einem Programm beliebig oft ***benutzt*** werden (z. B. um ein Unterprogramm aufzurufen). Bei jeder Benutzung muss der Ausführer entscheiden, ob diese Benutzung des Namens ***kein*** Unterprogramm bezeichnet

(Fehler), genau ***ein*** Unterprogramm bezeichnet (alles in Ordnung) oder ***mehr als ein*** Unterprogramm bezeichnet (Fehler).

Die genauen Regeln für die Namensauflösung in C++ sind ziemlich kompliziert ihre Darstellung im C++-Standard ist nicht ganz leicht zu verstehen. Ich vermute aber stark, dass beide verwendeten Compiler (Gnu-Cygnus-C++, Version 2.95.2 und Borland C++, Version 5.5) das Beispielprogramm ***Schablonen02*** nicht ganz korrekt behandeln: Der Gnu-Cygnus-Compiler erkennt nicht, dass der `put`-Aufruf in Zeile 38 mehrdeutig ist, und wenn man die Zeile 18 auskommentiert erkennt der Borland-Compiler nicht, dass dann der `put`-Aufruf in Zeile 37 mehrdeutig ist.

10.7 Das Beispielprogramm Schablonen03 (die Schablone druckeVector)

Schablonen schaffen manchmal Probleme, die sich nur mit Schablonen elegant lösen lassen. Z. B. kann man mit der Schablone `vector` sehr leicht viele `vector`-Klassen erzeugen. Für jede dieser Klassen braucht man dann aber häufig eine eigene Ausgabe-Prozedur (weil man mit einer Ausgabe-Prozedur für `vector<int>`-Objekten keine `vector<string>`-Objekte ausgeben kann etc.). Diese vielen Ausgabeprozeduren realisiert man am besten mit Hilfe einer ***Prozedurschablone***.

```
1   // Datei Schablonen03.cpp
2   /* ------------------------------------------------------------------------
3   Mit Hilfe der Prozedur-Schablone druckeVector kann man (die Komponenten
4   beliebiger) vector-Objekte ausgeben lassen. Die vector-Objekte koennen z. B.
5   zu den Klassen vector<char>, vector<int>, vector<string> etc. gehoeren.
6   Voraussetzung: Fuer die Komponenten der vector-Objekte muss ein Ausgabe-
7   Operator "<<" definiert sein.
8   ------------------------------------------------------------------------ */
9   #include <iostream>
10  #include <vector>
11  #include <string>
12  using namespace std;
13  // ------------------------------------------------------------------------
14  template<typename T>
15  void druckeVector(string name, const vector<T> &v) {
16     // Gibt name und v "in lesbarer Form" zur Standardausgabe aus.
17     // Der Typ-Parameter T dieser Prozedur-Schablone wird an die Klassen-
18     // Schablone vector "weitergereicht".
19     cout << name << " (" << v.size() << " Komp.): ";
20     for (vector<T>::size_type i=0; i<v.size(); i++) {
21        cout << v[i] << " ";
22     }
23     cout << endl;
24  } // druckeVector
25  // ------------------------------------------------------------------------
26  int main() {
27     // Reihungen (zum Initialisieren von vector-Objekten) vereinbaren:
28     char   rc[] = {'A', 'B', 'C', 'D', 'E'};
29     int    ri[] = {123, 456, 789};
30     string rs[] = {"Hallo", "Susi!", "Wie geht's?"};
31
32     // Vektoren verschiedener vector-Typen vereinbaren:
33     vector<char>   vc(rc, rc + sizeof(rc)/sizeof(rc[0]));
34     vector<int>    vi(ri, ri + sizeof(ri)/sizeof(ri[0]));
35     vector<string> vs(rs, rs + sizeof(rs)/sizeof(rs[0]));
36
37     // Die Vektoren (mit Instanzen der Schablone druckeVector) ausgeben:
38     druckeVector("vc", vc);
39     druckeVector("vi", vi);
40     druckeVector("vs", vs);
41  } // main
```

```
42  /* ------------------------------------------------------------------------
43  Ausgabe des Programms Schablonen03:
44
45  vc (5 Komp.): A B C D E
46  vi (3 Komp.): 123 456 789
47  vs (3 Komp.): Hallo Susi! Wie geht's?
48  ------------------------------------------------------------------------ */
```

10.8 Weitere Beispielprogramme mit Unterprogrammschablonen

Das Beispielprogramm `Schablonen05` enthält zwei Unterprogrammschablonen namens `Ganz-NachString` und `StringNachGanz`, mit deren Hilfe man ***Ganzzahlen*** in ***Strings*** bzw. ***Strings*** in ***Ganzzahlen*** umwandeln kann. Die Ganzzahlen können zu ***beliebigen Ganzzahltypen*** (`char`, `short`, `int`, ...) gehören.

Das Beispielprogramm `Schablonen07` enthält eine Unterprogrammschablone namens `summe`, mit der man Zahlen summieren kann, die sich in ***beliebigen Behälter-Objekten*** befinden (z. B. in `vec-tor`-Objekten oder in `list`-Objekten oder in `deque`-Objekten etc.).

10.9 Typparameter von Schablonen per Metaprogrammierung beschränken

Die Entwickler von C++ haben keine speziellen Konstrukte vorgesehen, mit denen man z. B. festlegen kann, dass der Typparameter `Ganz` der Schablone `hoch` (siehe oben den Abschnitte 10.5) nur durch ***Ganzzahltypen*** (aber nicht durch Bruchzahltypen wie `float` oder Klassentypen wie `string` etc.) ersetzt werden darf. Allerdings fand Erwin Unruh, ein Programmierer aus München, sozusagen innerhalb der Sprache C++ eine weitere Programmiersprache, die wir hier kurz als die **Sprache M** (wie Metaprogrammierung) bezeichnen werden. Bevor Herr Unruh 1994 (bei einer Sitzung des Komitees für die Standardisierung von C++) ein in M geschriebenes Programm vorlegte, ahnte wohl niemand, dass die Sprache C++ die Sprache M enthält.

Befehle der Sprache M haben die merkwürdige Eigenschaft, dass sie schon zur Compilezeit (und nicht erst später zur Laufzeit) ausgeführt werden. Mit Hilfe von M-Befehlen kann man z. B. prüfen, ob für den Parameter `Ganz` der Schablone `hoch` ein Ganzzahltyp angegeben wurde. Falls der Programmierer die Schablone mit einem falschen Typ (z. B. `float` oder `string` etc.) instanziiert hat, kann man vom Compiler eine Fehlermeldung ausgeben lassen. Zugegeben: Die Fehlermeldug ist nicht besonders leicht zu entziffern, aber vielleicht wird das mit der nächsten Version des C++-Standards besser.

Beispiel: Eine "sichere" Variante der Funktionsschablone `hoch`:

```
1   // Datei Hoch.hpp
2   #ifndef _HOCH_HPP_
3   #define _HOCH_HPP_
4
5   #include <boost/type_traits.hpp>    // fuer boost::is_integral<T>
6   #include <boost/static_assert.hpp>  // fuer BOOST_STATIC_ASSERT()
7
8   template<class Ganz>
9   Ganz hoch(Ganz basis, Ganz exponent) {
10
11     // Falls Ganz kein Ganzzahltyp ist, gibt der folgende
12     // Befehl (zur Compilezeit) eine Fehlermeldung aus:
13     BOOST_STATIC_ASSERT(boost::is_integral<Ganz>::value);
14
15     Ganz erg = 1;
16     for (Ganz i=1; i<=exponent; i++) erg *= basis;
17     return erg;
18  } // template gleich
19  #endif //_HOCH_HPP_
```

Im folgenden Programm `HochTst` werden zwei Instanzen der "sicheren" Funktionsschablone `hoch` aufgerufen, die falsche Instanz `hoch<double>` und die richtige `hoch<int>`. Die falsche Instanz wird vom Compiler mit einer Fehlermeldung abgelehnt (anders als oben im Abschnitt 10.5 im Programm `Schablonen01`):

```
1   // Datei HochTst.cpp
2   #include <iostream>   // cout, cin, <<, >>, endl
3   #include "Hoch.hpp"   // hoch
4   using namespace std;
5
6   int main() {
7      cout << "HochTst: Jetzt geht es los!"        << endl;
8   // cout << "hoch(1.5, 2.5): " << hoch(1.5, 2.5) << endl; // falsch
9      cout << "hoch(  2,  10): " << hoch(  2,  10) << endl; // richtig
10     cout << "HochTst: Das war's erstmal!"        << endl;
11  } // main
12  /* ------------------------------------------------------------------------
13  Fehlermeldung des Gnu-C++-Compilers, wenn die entsprechende Zeile kein
14  Kommentar ist:
15
16  Hoch.hpp: In function `Ganz hoch(Ganz, Ganz) [with Ganz = double]':
17  HochTst.cpp:8:   instantiated from here
18  Hoch.hpp:13: error: incomplete type `boost::STATIC_ASSERTION_FAILURE<false>'
19     does not have member `value'
20  ------------------------------------------------------------------------
21  Ausgabe des Programms HochTst:
22
23  HochTst: Jetzt geht es los!
24  hoch(  2,  10): 1024
25  HochTst: Das war's erstmal!
26  ------------------------------------------------------------------------ */
```

Eine Gruppe von Programmierern ("die Boost-Gruppe") hat viele nützliche M-Befehle (z. B. `is_integral`, `is_float` etc. und `BOOST_STATIC_ASSERT` etc.) entwickelt und stellt sie als sogenannten Boost-Bibliothek (siehe `www.boost.org`) der Allgemeinheit zur Verfügung.

Wenn diese Boost-Bibliothek (boost library) im Verzeichnis `d:\bibs\boost` steht, kann man in einem Cygwin-bash-Fenster die Quelldatei `HochTst.cpp` mit einem der folgenden Kommandos compilieren und zu einer ausführbaren Datei namens `HochTst.exe` binden (mit dem Gnu-C++-Compiler `g++`, dem Borland-C++-Kommandozeilencompiler `bcc32` bzw. dem Microsoft-C++-Compiler `cl`):

```
> g++   -o HochTst.exe -I/cygdrive/d/bibs/boost HochTst.cpp
> bcc32                -I/cygdrive/d/bibs/boost HochTst.cpp
> cl             /EHsc /I         D:/bibs/boost HochTst.cpp
```

Wenn man den Pfadnamen der Boost-Bibliothek in eine Umgebungsvariablen namens `BOOST_ROOT` geschrieben hat, kann man auch das folgende, kürzere Kommando verwenden, um die Quelldatei `HochTst.cpp` zu compilieren und zu binden:

```
> g++ -o HochTst.exe -I$BOOST_ROOT HochTst.cpp
```

Anmerkung: Mit den Befehlen der Sprache M ("per Metaprogrammierung") kann man wesentlich mehr machen, als nur die Typparameter von Schablonen prüfen und einschränken. Informationen dazu findet man auf der Netzseite der Boost-Organisation (`www.boost.org`).

Die Sprache M besteht vor allem aus ***den*** C++-Befehlen, mit denen man Klassenschablonen (siehe unten den Abschnitt 13) vereinbaren und benutzen kann. Diese Klassenschablonen werden in der Sprache M aber auf eine ganz andere Weise benutzt und (vom Programmierer) interpretiert als in normalen C++-Programmen, nämlich als sog, Meta-Funktionen. Außerdem ist die Sprache M keine prozedurale

Sprache wie C++, sondern ein ***funktionale*** Sprache, d. h. es gibt in M keine veränderbaren Variablen und keine Zuweisung.

Die Metaprogrammierung und die Boost-Bibliothek sind auch unter C++-Programmierern noch nicht allgemein bekannt. Gerade alte C++-Hasen kann man deshalb häufig damit beeindrucken, dass man ganz nebenbei seine Vertrautheit mit der Metaprogrammierung erwähnt.

11 Ein- und Ausgabe mit Strömen

Eine ***Datenquelle*** ist irgendein Ding, von dem ein Programm Daten ***holen*** kann, z. B. eine Tastatur, eine Datei auf einer Festplatte, eine String-Variable im Hauptspeicher, eine Verbindung zum Internet etc. Eine ***Datensenke*** ist irgendein Ding, zu dem ein Programm Daten ***schicken*** kann, z. B. ein Bildschirm, ein Drucker, eine Datei auf einer Festplatte, eine String-Variable im Hauptspeicher, eine Verbindung zum Internet etc.

Ein ***Eingabestrom*** ist eine ***Verbindung*** zwischen einer ***Datenquelle*** und einem ***Programm***. Verschiedene Arten von Datenquellen erfordern verschiedene Eingabeströme (z. B. erfordern ***Dateien*** andere Ströme als eine ***Verbindung zum Internet*** oder zu einer ***Tastatur***). Auf der anderen Seite (zum Programm hin) bieten die verschiedenen Eingabeströme aber eine ***einheitliche*** Schnittstelle. So helfen Eingabeströme, von den Unterschieden zwischen verschiedenen Datenquellen zu ***abstrahieren***, so dass das Programm nicht (oder nur geringfügig) geändert werden muss, wenn man eine Datenquelle gegen eine andere austauscht. Ganz entsprechendes gilt für ***Ausgabeströme*** und ***Datensenken***.

Ströme werden in mehreren Programmiersprachen (darunter auch C++ und Java) als ***Objekte*** entsprechender Strom-Klassen realisiert. Ein ***Eingabestrom-Objekt*** muss vom Programmierer vereinbart und mit einer ***Datenquelle*** verbunden werden (z. B. mit einer Datei oder mit einer Tastatur etc.). Das Eingabestrom-Objekt enthält dann Methoden, mit denen man Daten (direkt aus dem Strom, indirekt aus der Datenquelle) ***einlesen*** kann.

Ein ***Ausgabestrom-Objekt*** muss vereinbart und mit einer ***Datensenke*** verbunden werden (z. B. mit einer Datei oder einem Bildschirm etc.). Das Ausgabestrom-Objekt enthält dann Methoden, mit denen man Daten (direkt in den Strom, indirekt in die Datensenke) ***schreiben*** kann.

Unter einem ***Dateistrom*** (file stream) versteht man einen ***Strom***, der speziell mit einer ***Datei*** (und nicht mit einem Gerät wie Tastatur oder Bildschirm oder einer anderen Datenquelle bzw. -senke) verbunden werden kann.

Die C++-Standardbibliothek stellt dem Programmierer eine Reihe von ***Strom-Klassen*** zur Verfügung, darunter die Klassen ***istream***, ***ostream*** und ***iostream*** für allgemeine Ströme und ***ifstream***, ***ofstream*** und ***fstream*** speziell für Dateiströme. Außerdem kann man in jedem C++-Programm die folgenden 4 ***Strom-Objekte*** benutzen, ohne sie vereinbaren oder mit einer Datenquelle bzw. -senke verbinden zu müssen:

`cin`	Standardeingabe	(ein `istream`-Objekt)
`cout`	Standardausgabe	(ein `ostream`-Objekt)
`cerr`	Standardfehlerausgabe	(ein `ostream`-Objekt)
`clog`	Standardprotokollausgabe	(ein `ostream`-Objekt)

Das verwendete ***Betriebssystem*** verbindet diese Strom-Objekte automatisch mit bestimmten Datenquellen bzw. -senken. Je nach Betriebssystem hat der Benutzer mehr oder weniger Möglichkeiten, diese Verbindungen durch ***Dateiumlenkungen*** zu verändern (z. B. kann man unter Dos/Windows Ausgaben nach `cerr` und `clog` leider ***nicht*** umlenken, während das unter Unix-Betriebssystemen natürlich möglich ist). Üblicherweise sind `cout`, `cerr` und `clog` mit einem ***Bildschirm*** und `cin` mit einer ***Tastatur*** verbunden.

Hier die wichtigsten Tatsachen über C++-Ströme in Stichworten:

1. Die vom C++-Standard vorgeschriebenen ***Ströme*** (streams) sind sehr ***maschinennah*** und ***simpel***. Ein Strom ist eine ***Folge von Bytes***. In einen ***Ausgabestrom*** kann man Bytes hineinschreiben, aus einem ***Eingabestrom*** kann man Bytes herauslesen.

2. Typprüfungen finden bei der Ein-/Ausgabe ***kaum*** statt. Der Programmierer ist weitgehend dafür verantwortlich, seine getypten Werte in Bytefolgen bzw. Bytefolgen in getypte Werte umzuwandeln. Die C++-E/A-Befehle sind somit sehr ***schnell*** und ziemlich ***unsicher*** (d. h. sie bieten dem Programmierer zahlreiche Möglichkeiten, Fehler zu machen, die nicht automatisch entdeckt werden).

3. Die Kopfdatei *<**iostream**>* enthält Deklarationen der 8 Standardströme (`cin`, `cout`, `cerr`, `clog` für `char`-Werte und `wcin`, `wcout`, `wcerr`, `wclog` für "breite" Zeichen des Typs `wchar_t`).

4. So genannte Manipulatoren dienen dazu, Daten beim Ausgeben zu formatieren bzw. formatierte Daten einzulesen. So kann man z. B. Ganzzahlen in verschiedenen Darstellungen ausgeben und einlesen (dezimal, oktal, hexadezimal) und die Darstellung von Gleitpunktzahlen beeinflussen (Anzahl der Stellen nach dem Dezimalpunkt, Exponent oder nicht etc.). Dazu braucht man die Kopfdatei `<iomanip>`.

5. Will man selbst Strom-Objekte vereinbaren und mit Dateien verbinden, braucht man die Kopfdatei `<fstream>`.

6. Strom-Objekte haben einen inneren ***Zustand*** ("ein Gedächtnis"). Wenn z. B. beim Einlesen von `cin` ein ***Formatfehler*** auftritt (der Benutzer hat z. B. Buchstaben statt Ziffern eingegeben), dann wird im `cin`-Objekt ein entsprechendes Zustandsbit auf 1 gesetzt. Ehe man dieses Bit nicht wieder auf 0 gesetzt hat (z. B. mit dem Befehl `cin.clear()`) kann man ***keine weiteren Daten*** von `cin` einlesen. Entsprechendes gilt auch für andere Ströme.

7. Normalerweise besitzt jeder Strom einen internen ***Puffer***. In einem ***Ausgabepuffer*** werden Ausgabedaten gesammelt, bevor sie physikalisch ausgegeben werden. In einen ***Eingabepuffer*** werden häufig ***mehr*** Daten gelesen, als der Programmierer aktuell verlangt hat. Durch solche Pufferung wird die Ein-/Ausgabe in der Regel erheblich ***beschleunigt***. Nachteil 1: Datensätze werden evtl. nicht sofort auf eine Platte geschrieben und sind dann im Falle eines ***Rechnerabsturzes*** verloren. Nachteil 2: bei der Ausgabe zum ***Bildschirm*** werden die ausgegeben Daten unter Umständen ***nicht sofort sichtbar***. Mit der `flush`-Methode kann man dem Ausführer befehlen, alle Daten aus einem Ausgabepuffer "jetzt sofort" auszugeben (z. B. `cout.flush()`).

8. Man kann einen Eingabestrom `E` mit einem Ausgabestrom `A` ***verknüpfen*** (mit der Methode `tie`). Das bedeutet: Wenn man von `E` etwas einliest wird vorher der Befehl `A.flush();` ausgeführt, d. h. es werden alle im Ausgabepuffer für `A` liegenden Daten ausgegeben. Beispiel: Der Eingabestrom `cin` ist mit dem Ausgabestrom `cout` in dieser Weise verbunden.

```
1       cin.tie(NULL);   // Die Verknuepfung zwischen cin und cout aufheben.
2       ...
3       cin.tie(&cout);  // cin und cout miteinander verknuepfen.
4       ...
5       ... cin.tie()... // Liefert den mit cin verknuepften Strom (hier: &cout)
```

11.1 Formatierte und unformatierte Ein-/Ausgabe

Zu einem ***Bildschirm*** kann man eigentlich nur ***Zeichen*** und ***Zeichenketten*** (z. B. Werte der Typen ***char*** und ***char[]***) ausgeben. Andere Daten (z. B. ***int***-Werte, ***float***-Werte etc.) müssen vor der Ausgabe zu einem Bildschirm von ihrer ***internen Darstellung*** in eine ***externe Darstellung*** als Zeichenketten umgewandet werden. Entsprechend kann man von einer ***Tastatur*** eigentlich nur ***Zeichen*** und ***Zeichenketten*** einlesen. Andere Werte müssen in Form von Zeichenketten eingelesen und dann von ihrer ***externen Darstellung*** in eine ***interne Darstellung*** umgewandelt werden.

Ein-/Ausgaben, bei denen die Daten derart ***umgewandelt*** werden, bezeichnet man auch als ***Ein-/Ausgaben in einer von Menschen lesbaren Form*** oder kürzer als ***formatierte Ein-/Ausgaben*** oder als ***Text-Ein-/Ausgabe***.
Wenn man Daten in eine ***Datei*** (z. B. auf einer Festplatte) ausgibt oder sie von dort einliest, braucht man sie im allgemeinen nicht umzuwandeln (weil nur wenige Menschen die Daten auf einer Festplatte direkt mit einem Mikroskop und ohne Hilfe eines geeigneten Programms lesen). Man spricht dann von ***binärer*** oder von ***unformatierter Ein-/Ausgabe***.

In C++ kann man Daten mit Hilfe von ***Strömen*** sowohl ***formatiert*** als auch ***unformatiert*** ein- und ausgeben. Zum ***formatierten*** Ein-/Ausgeben benützt man meistens die ***Operatoren*** >> (Eingabe) und << (Ausgabe). Für die ***unformatierte*** Ein-/Ausgabe gibt es die Prozeduren ***read*** und ***write***. Die folgenden beiden Beispielprogramme sollen zeigen, wie man Daten formatiert bzw. unformatiert ein- und ausgeben kann.

```
1   // Datei EinAus10.cpp
2   /* ------------------------------------------------------------------------
3   Ein paar Ganzzahlen werden in einer von Menschen lesbaren Form in eine
4   Datei geschrieben und von dort wieder eingelesen. Meldungen werden nach
5   cout, Protokolldaten nach clog und Fehlermeldungen nach cerr ausgegeben.
6   ------------------------------------------------------------------------ */
7   #include <fstream>  // fuer ofstream, ifstream, cout, cerr, clog
8   #include <string>
9   #include <vector>
10  using namespace std;
11  // ------------------------------------------------------------------------
12  int main() {
13     cout << "EinAus10: Jetzt geht es los!" << endl;
14
15     int r[] = {87, 73, 69, 83, 79};   // Daten zum Schreiben in eine Datei
16     string dateiName("EinAus10.dat"); // Name der Datei
17     // ---------------------------------------------------------------------
18     // Einen Ausgabestrom erzeugen
19     // (aber noch nicht mit einer Datei verbinden):
20     ofstream aStrom;
21
22     // Den Ausgabestrom mit einer Datei verbinden:
23     aStrom.open(dateiName.c_str(), ios::binary | ios::out);
24
25     if (!aStrom) { // Wenn das Verbinden nicht geklappt hat:
26        cerr << "Konnte Ausgabedatei " << dateiName  // Fehler-
27             << " nicht oeffnen!"      << endl;      // meldung und
28        exit(-1);                                    // Programmabbruch
29     }
30
31     // Daten (int-Werte) in die Datei ausgeben:
32     for (size_t i=0; i<sizeof(r)/sizeof(r[0]); i++) {
33        aStrom << r[i] << " "; // Mindestens ein transparentes Zeichen (white
34                               // space) zum Trennen muss sein!
35     }
36     aStrom.close(); // Verbindung zwischen Ausgabestrom und Datei aufheben
```

```
37     // ------------------------------------------------------------------
38     // Ausgegebene Daten protokollieren:
39     clog << "In  die Datei " << dateiName << " wurden geschrieben: ";
40     for (size_t i=0; i<sizeof(r)/sizeof(r[0]); i++) {
41        clog << r[i] << " ";
42     }
43     clog << endl;
44     // ------------------------------------------------------------------
45     // Einen Eingabestrom erzeugen und sofort mit einer Datei verbinden:
46     ifstream eStrom(dateiName.c_str(), ios::binary | ios::in);
47
48     if (!eStrom) { // Wenn das Verbinden nicht geklappt hat:
49        cerr << "Konnte Eingabedatei " << dateiName  // Fehler-
50             << " nicht oeffnen!"      << endl;      // meldung und
51        exit(-1);                                    // Programmabbruch
52     }
53     // ------------------------------------------------------------------
54     // Daten (int-Werte) aus der Datei lesen und in einem Vector speichern:
55     int         zahl; // Zum Einlesen eines int-Wertes
56     vector<int> v;    // Ein Vector kann mit seiner Aufgabe wachsen!
57
58     while (true) {
59        eStrom  >> zahl;
60        if (eStrom.eof()) break;
61        v.push_back(zahl);
62     }
63     eStrom.close(); // Verbindung zwischen Eingabestrom und Datei aufheben.
64     // ------------------------------------------------------------------
65     // Eingelesene Daten protokollieren:
66     clog << "Aus der Datei " << dateiName << " wurden gelesen:     ";
67     for (size_t i=0; i<v.size(); i++) {
68        clog << v[i] << " ";
69     }
70     clog << endl;
71     // ------------------------------------------------------------------
72     cout << "EinAus10: Das war's erstmal!" << endl;
73  } // main
74  /* ---------------------------------------------------------------------
75  Ausgabe des Programms EinAus10 zum Bildschirm:
76
77  EinAus10: Jetzt geht es los!
78  In  die Datei EinAus10.dat wurden geschrieben: 87 73 69 83 79
79  Aus der Datei EinAus10.dat wurden gelesen:     87 73 69 83 79
80  EinAus10: Das war's erstmal!
81  ---------------------------------------------------------------------
82  Inhalt der Datei EinAus10.dat in hexadezimaler Darstellung:
83
84  000000  38 37 20 37  33 20 36 39  20 38 33 20  37 39 20     87 73 69 83 79
85  ------------------------------------------------------------------- */
```

Das folgende Programm unterscheidet sich syntaktisch ***nur geringfügig*** vom vorigen Programm, leistet aber etwas ***ziemlich anderes***:

```
1   // Datei EinAus11.cpp
2   /* ---------------------------------------------------------------------
3   Ein paar Ganzzahlen werden in ihrer internen Darstellung ("binaer") in eine
4   Datei geschrieben und von dort wieder eingelesen. Meldungen werden nach
5   cout, Protokolldaten nach clog und Fehlermeldungen nach cerr ausgegeben.
6   ------------------------------------------------------------------- */
7   #include <fstream>  // fuer ofstream, ifstream, cout, cerr, clog
8   #include <string>
9   #include <vector>
10  using namespace std;
```

```
11 // ------------------------------------------------------------------------
12 int main() {
13    cout << "EinAus11: Jetzt geht es los!" << endl;
14
15    int r[] = {87, 73, 69, 83, 79};   // Daten zum Schreiben in eine Datei
16    string dateiName("EinAus11.dat"); // Name der Datei
17    // ---------------------------------------------------------------------
18    // Einen Ausgabestrom erzeugen und sofort mit einer Datei verbinden:
19    ofstream aStrom(dateiName.c_str(), ios::binary | ios::out);
20
21    if (!aStrom) { // Wenn das Verbinden nicht geklappt hat:
22       cerr << "Konnte Ausgabedatei " << dateiName  // Fehler-
23            << " nicht oeffnen!"      << endl;      // meldung und
24       exit(-1);                                    // Programmabbruch
25    }
26
27    // Daten (int-Werte) in die Datei ausgeben:
28    for (size_t i=0; i<sizeof(r)/sizeof(r[0]); i++) {
29       aStrom.write(reinterpret_cast<const char *>(&r[i]), sizeof(r[i]));
30    }
31    aStrom.close(); // Verbindung zwischen Ausgabestrom und Datei aufheben
32    // ---------------------------------------------------------------------
33    // Ausgegebene Daten protokollieren:
34    clog << "In  die Datei " << dateiName << " wurden geschrieben: ";
35    for (size_t i=0; i<sizeof(r)/sizeof(r[0]); i++) {
36       clog << r[i] << " ";
37    }
38    clog << endl;
39    // ---------------------------------------------------------------------
40    // Einen Eingabestrom erzeugen
41    // (aber noch nicht mit einer Datei verbinden):
42    ifstream eStrom;
43
44    // Den Eingabestrom mit einer Datei verbinden:
45    eStrom.open(dateiName.c_str(), ios::binary | ios::in);
46
47    if (!eStrom) { // Wenn das Verbinden nicht geklappt hat:
48       cerr << "Konnte Eingabedatei " << dateiName  // Fehler-
49            << " nicht oeffnen!"      << endl;      // meldung und
50       exit(-1);                                    // Programmabbruch
51    }
52    // ---------------------------------------------------------------------
53    // Daten (int-Werte) aus der Datei lesen und in einem Vector speichern:
54    int         zahl; // Zum Einlesen eines int-Wertes
55    vector<int> v;    // Ein Vector kann mit seiner Aufgabe wachsen!
56
57    while (true) {
58       eStrom.read(reinterpret_cast<char *>(&zahl), sizeof(zahl));
59       if (eStrom.eof()) break;
60       v.push_back(zahl);
61    }
62    eStrom.close(); // Verbindung zwischen Eingabestrom und Datei aufheben.
63    // ---------------------------------------------------------------------
64    // Eingelesene Daten protokollieren:
65    clog << "Aus der Datei " << dateiName << " wurden gelesen:     ";
66    for (size_t i=0; i<v.size(); i++) {
67       clog << v[i] << " ";
68    }
69    clog << endl;
70    // ---------------------------------------------------------------------
71    cout << "EinAus11: Das war's erstmal!" << endl;
72 } // main
```

```
73  /* ------------------------------------------------------------------------
74  Ausgabe des Programms EinAus11 zum Bildschirm:
75
76  EinAus11: Jetzt geht es los!
77  In  die Datei EinAus11.dat wurden geschrieben: 87 73 69 83 79
78  Aus der Datei EinAus11.dat wurden gelesen:     87 73 69 83 79
79  EinAus11: Das war's erstmal!
80  ------------------------------------------------------------------------
81  Inhalt der Datei EinAus11.dat in hexadezimaler Darstellung:
82
83  000000  57 00 00 00  49 00 00 00  45 00 00 00  53 00 00 00  W...I...E...S...
84  000010  4F 00 00 00                                         O...
85
86  Wieso steht da ganz rechts W...I...E...S...O...?
87  ---------------------------------------------------------------------- */
```

Die unformatierte Ein-/Ausgabe erfolgt unter Ausschaltung aller Typprüfungen. Deshalb ist sie schnell und fehlerträchtig. Der write-Befehl (Zeile 29) erwartet nur eine Adresse (von `char`), ab der Bytes aus dem Hauptspeicher ausgegeben werden sollen, und die Anzahl der auszugebenden Bytes. Der genaue Typ der auszugebenden Daten interessiert nicht und muss mit dem Befehl `reinterpret_cast` beseitigt werden (nur beim Ausgeben von Daten des Typs `char` kann diese Typumwandlung entfallen). Ganz entsprechend erwartet der `read`-Befehl (Zeile 58) nur eine Adresse, ab der die eingelesenen Bytes im Hauptspeicher abgelegt werden sollen, und die Anzahl der einzulesenden Bytes. Auch hier müssen alle störenden Typinformationen mit dem Befehl `reinterpret_cast` beseitigt werden.

In der Datei `EinAus11.dat` (die vom Programm `EinAus11` erstellt wird) stehen die ausgegebenen ***int***-Werte in Form von Bitketten ***ohne*** jegliche Informationen über ihren ***Typ***. Das Programm `Ein-Aus12` (hier nicht wiedergegeben) liest diese Daten ein und protokolliert sie nach `clog` ganz ähnlich wie das Programm ***EinAus11***, nur mit dem folgendem kleinen Unterschied: Die Variable `zahl` (in `EinAus11.cpp` vereinbart in Zeile ***54***) ist nicht vom Typ `int`, sondern vom Typ `float` und der Vector `v` (vereinbart in Zeile ***55***) ist entsprechend nicht vom Typ `vector<int>` sondern vom Typ `vector<float>. Hier` die Ausgabe des Programms `EinAus12`:

```
1   Aus der Datei EinAus11.dat wurden gelesen:
2   1.21913e-43 1.02295e-43 9.66896e-44 1.16308e-43 1.10703e-43
```

Die Bitketten, die das Programm `EinAus11` als Codes der fünf `int`-Werte `87`, `73`, `69`, `83` und `79` in die Datei geschrieben hat, wurden vom Programm `EinAus12` als Codes für `float`-Werte eingelesen und interpretiert. Dieser Fehler wird in C++ durch ***keine*** Typprüfung aufgedeckt.

11.2 Verschiedene Befehle zum Einlesen von Daten

In vielen Fällen ist das Einlesen von Daten deutlich ***schwieriger*** als das ***Ausgeben***, weil man beim Einlesen oft nicht weiß, "was von draussen reinkommt" und entsprechend "auf alles gefasst sein" muss. Deshalb gibt es auch in C++ zahlreiche Befehlen zum Einlesen von Daten, die sich zum Teil nur subtil voneinander unterscheiden. Das folgend Beispielprogramm demonstriert ***drei*** davon.

```
1   // Datei EinAus13.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert den Unterschied zwischen den Einlesebefehlen cin.get(c),
4   getline(cin, s) und cin >> s; (wobei c ein char und s ein string ist).
5   ---------------------------------------------------------------------- */
6   #include <iostream>
7   #include <string>
8   using namespace std;
9   // ------------------------------------------------------------------------
```

```
10  int main() {
11     cout << "EinAus13: Jetzt geht es los!"                        << endl;
12     char     c1, c2;
13     string   s1, s2,  s3;
14
15     cout << "Bitte ein Zeichen und Return eingeben: ";
16     cin.get(c1);
17     cin.get(c2);
18
19     cout << "Sie haben 2 Zeichen eingegeben!        "             << endl;
20     cout << "Zeichen 1 als char ausgegeben        : " <<       c1 << endl;
21     cout << "Zeichen 2 als int  ausgegeben        : " << (int) c2 << endl;
22
23     cout << "Bitte eine Zeichenkette und Return   : ";
24     getline(cin, s1);
25     cout << "Ihre Eingabe war                     : " <<       s1 << endl;
26
27     cout << "Bitte eine Zeichenkette und Return   : ";
28     cin  >> s2;
29     cout << "Ihre Eingabe war                     : " <<       s2 << endl;
30     getline(cin, s3);
31     cout << "Oh, im Tastaturpuffer stand noch     : " <<       s3 << endl;
32     cout << "(" << s3.size() << " Zeichen)          "             << endl;
33
34     cout << "EinAus13: Das war's erstmal!"                        << endl;
35  } // main
36  /* ------------------------------------------------------------------------
37  Ausgabe des Programms EinAus13, compiliert mit Borland C++, Version 5.5:
38
39  EinAus13: Jetzt geht es los!
40  Bitte ein Zeichen und Return eingeben: X
41  Sie haben 2 Zeichen eingegeben!
42  Zeichen 1 als char ausgegeben        : X
43  Zeichen 2 als int  ausgegeben        : 10
44  Bitte eine Zeichenkette und Return   :   Hallo Susi!
45  Ihre Eingabe war                     :   Hallo Susi!
46  Bitte eine Zeichenkette und Return   :   How are you?
47  Ihre Eingabe war                     : How
48  Oh, im Tastaturpuffer stand noch     : are you?
49  (8 Zeichen)
50  EinAus13: Das war's erstmal!
51  ------------------------------------------------------------------------
52  Ausgabe des Programms EinAus13, compiliert mit Gnu-Cygnus, Version 2.95.2,
53  bei gleicher Eingabe, nur die letzten 5 Zeilen:
54
55  ...
56  Bitte eine Zeichenkette und Return   :   How are you?
57  Ihre Eingabe war                     : How
58  Oh, im Tastaturpuffer stand noch     :  are you?
59  (9 Zeichen)
60  EinAus13: Das war's erstmal!
61  --------------------------------------------------------------------- */
```

Anstelle der Standardeingabe `cin` hätte man in diesem Programm auch irgendeinen anderen Eingabestrom nehmen können, z. B. einen ***Dateistrom***, der mit einer ***Eingabedatei*** verbunden ist.

Der Befehl `cin.get(c1)` liest genau ein Zeichen aus dem Strom, auch Steuerzeichen wie `LF` (line feed, dez `10`, hex `A`) und `CR` (carriage return, dez `13`, hex `D`) und transparente Zeichen (white space characters) wie ***Blank*** (dez 32, hex 20) oder ***Tab*** (horizontal tab, dez `9`, hex `9`). Wenn der Benutzer z.

B. Buchstaben X eingibt und dann auf die Return-Taste drückt, kommen im Programm zwei Zeichen an: das X und LF-Zeichen (ein CR-Zeichen wird vom Dos/Windows-Betriebssystem verschluckt).

Der Befehl getline(cin, s1) liest alle Zeichen bis zur nächsten ***Zeilenende-Markierung*** in den String s. Die Zeilenende-Markierung besteht je nach Betriebssystem aus einem LF-Zeichen (Unix), einem CR-Zeichen (Macintosh) bzw. einem CR- gefolgt von einem LF-Zeichen (Dos/Windows). Die Zeilenende-Markierung selbst wird zwar aus dem Eingabestrom entfernt, sollte aber nicht in den String s übertragen werden. Tatsächlich wird beim Borland- und beim Gnu-Cygnus-Compiler unter Windows das CR-Zeichen nach s übertragen und nur das LF-Zeichen weggeworfen. Beim Versuch, portable Programme zu schreiben oder Programme zu portieren (von einer Plattform auf eine andere zu übertragen) sitzt der Teufel in vielen solchen Details.

Der Befehl cin >> s2; veranlasst den Ausführer zu folgenden Aktionen:

1. Er liest aus dem Eingabestrom cin alle ***transparenten Zeichen*** (Blank, Tab, LF und CR) und wirft sie weg.

2. Dann liest er alle ***nicht-transparenten*** Zeichen nach s2 bis in cin wieder ein transparentes Zeichen kommt. Dieses abschliessende transparente Zeichen sollte er nicht aus dem Eingabestrom entfernt. Der Gnu-Cygnus-Ausführer hält sich scheints an diese Regel, der Borland-Ausführer offenbar nicht, zumindest nicht immer (siehe oben Ausgabe des Programms EinAus13).

Im folgenden Beispielprogramm wird eine Textdatei ***zeilenweise*** gelesen und die ***Leseposition*** mit dem Befehl seekg manipuliert:

```
1   // Datei Einlesen04.cpp
2   // ------------------------------------------------------------------------
3   // Alle Zeilen einer Datei einlesen mit getline, das Ende der Datei er-
4   // kennen (auf zwei verschiedene Weisen) und die Leseposition in einer
5   // Eingabedatei repositionieren (mit seekg).
6   // ------------------------------------------------------------------------
7   #include <fstream>  // fuer ifstream und cout
8   #include <string>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  int main() {
12     string   pfad  = "test.txt";     // Pfad der Eingabedatei
13     string   einZeile;
14     ifstream einDatei(pfad.c_str(), ios::in | ios::binary);
15
16     // Falls das Oeffnen der Datei nicht geklappt hat:
17     if (!einDatei) {
18        cout << "Die Datei " << pfad <<
19                "  konnte nicht geoeffnet werden!"     << endl;
20        return -1;
21     }
22
23     // Alle Zeilen der Datei lesen und ausgeben:
24     cout << "AAAAAAAAAAAAAA" << endl;
25     while (!einDatei.eof()) {        // Wenn noch nicht end of file
26         getline(einDatei, einZeile); // Eine Zeile einlesen und
27         cout << einZeile << endl;    // zur Standardausgabe ausgeben.
28     }
29
30     cout << "BBBBBBBBBBBBBB" << endl;
31
32     // Die Leseposition an den Anfang der Datei positionieren:
33     einDatei.clear();                // eof-Bit ausknipsen
34     einDatei.seekg(0, ios::beg);     // Byte 0 relativ zum Datei-beg-inn
35
```

```
36     // Nochmal alle Zeilen der Datei lesen und ausgeben:
37     while (getline(einDatei, einZeile)) { // Wenn noch eine Zeile da war
38        cout << einZeile << endl;          // zur Standardausgabe ausgeben.
39     }
40
41     cout << "CCCCCCCCCCCCCC" << endl;
42     einDatei.close();
43  } // main
44  /* ------------------------------------------------------------------------
45  Die Ausgabe des Programms Einlesen04 sieht auf dem Bildschirm so aus:
46
47  AAAAAAAAAAAAAA
48  11111
49  22222
50  33333
51  444444
52  BBBBBBBBBBBBBB
53  11111
54  22222
55  33333
56  444444
57  CCCCCCCCCCCCCC
58  ------------------------------------------------------------------------
59  In hexadezimaler Darstellung ("in Wirklichkeit") sieht die Ausgabe so aus:
60
61  000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 0D 0A  AAAAAAAAAAAAAA..
62  000010  31 31 31 31  31 0D 0D 0A  32 32 32 32  32 0D 0D 0A  11111...22222...
63  000020  33 33 33 33  33 0D 0D 0A  34 34 34 34  34 34 0D 0A  33333...444444..
64  000030  42 42 42 42  42 42 42 42  42 42 42 42  42 42 0D 0A  BBBBBBBBBBBBBB..
65  000040  31 31 31 31  31 0D 0D 0A  32 32 32 32  32 0D 0D 0A  11111...22222...
66  000050  33 33 33 33  33 0D 0D 0A  34 34 34 34  34 34 0D 0A  33333...444444..
67  000060  43 43 43 43  43 43 43 43  43 43 43 43  43 43 0D 0A  CCCCCCCCCCCCCC..
68  ---------------------------------------------------------------------- */
```

11.3 Eingabefehler beim formatierten Einlesen abfangen und behandeln

Sei ***n*** eine ***int***-Variable. Dann veranlasst der Befehl ***cin >> n;*** den Ausführer zu folgenden Aktionen:

1. Er liest aus dem Eingabestrom ***cin*** alle ***transparenten Zeichen*** (Blank, Tab, `LF` und `CR`) und wirft sie weg.

2. Wenn dann ein ***falsches*** Zeichen kommt (***richtig*** sind nur die ***Vorzeichen*** `'+'` und `'-'` und die ***Ziffern*** `'0'` bis `'9'`) entfernt er dieses Zeichen ***nicht*** aus `cin` und verändert auch die Variable `n` ***nicht***. Statt dessen setzt er im Eingabestrom `cin` ein Fehlerbit auf `true`. Bevor dieses Fehlerbit nicht wieder auf `false` zurückgesetzt wurde, ***scheitert*** jeder Versuch, aus `cin` (mit ***irgendeinem Befehl***) weitere Daten zu lesen.

3. Wenn statt eines falschen Zeichens ***richtige Zeichen*** kommen (z. B. `"345..."` oder `"+8765..."` oder `"-5..."` etc.), liest er möglichst viele richtige Zeichen aus `cin` (bis zum nächsten ***falschen*** oder ***transparenten*** Zeichen), wandelt die Kette dieser richtigen Zeichen in einen `int`-Wert um und weist der Variablen `n` diesen Wert zu. Die richtigen Zeichen werden alle aus `cin` entfernt.

Mit dem folgenden Beispielprogramm kann mit diese etwas abstrakten Regeln konkret erproben:

```
1   // Datei EinAus01.cpp
2   /* ------------------------------------------------------------------------
3   Eine robuste Routine zum Einlesen von Ganzzahlen von der Standardeingabe.
4   Wenn der Benutzer falsche Daten eingibt, wird der Eingabepuffer geloescht
5   und die Fehlerbits im Strom cin werden zurückgesetzt.
6   ------------------------------------------------------------------------ */
7   #include <iostream> // fuer Standardausgabe cout und Standardeingabe cin
8   #include <string>   // fuer den Typ string (gehoert nicht zum Sprachkern!)
9   #include <iomanip>  // fuer setw (set width, Breite einer Ausgabe)
10  using namespace std;
11
12  int main() {
13     string muell;    // Zum Einlesen "falscher Zeichen"
14     int    g1, g2;   // Zum Einlesen zweier Ganzzahlen
15
16     cout << "EinAus01: Jetzt geht es los!" << endl;
17
18     // Solange von cin lesen, bis der Benutzer 2 korrekte Ganzzahlen
19     // eingibt, bei Falscheingabe Fehlermeldung:
20     while (true) {             // Schleife wird mit break verlassen
21        cout << "Zwei Ganzzahlen?     ";
22        cin  >>  g1 >> g2;      // Versuch, zwei Ganzzahlen zu lesen.
23
24        if (!cin.fail()) break; // Falls der Versuch nicht missglueckt ist
25
26        // Falls der Versuch missglueckt ist (Formatfehler):
27        cin.clear();            // Fehlerbit(s) in cin ausschalten
28        getline(cin, muell);    // Falsche Zeichen bis Zeilenende lesen
29        int len = muell.size(); // Anzahl der falschen Zeichen nach len
30        cout << setw(3) << len << " falsche Zeichen: " << muell << endl;
31     } // while
32     cout << "Ihre Eingabe:         " << g1 << " und " << g2 << endl;
33  } // main
34  /* ------------------------------------------------------------------------
35  Ein Dialog mit dem Programm EinAus01:
36
37  EinAus01: Jetzt geht es los!
38  Zwei Ganzzahlen?     abc +456
39    7 falsche Zeichen: abc 456
40  Zwei Ganzzahlen?     -123 abc
```

```
41    3 falsche Zeichen: abc
42  Zwei Ganzzahlen?     -123 +456DM10Pfennige
43  Ihre Eingabe:        -123 und 456
44  ------------------------------------------------------------------------ */
```

Der Befehl `getline(cin, muell);` (in Zeile 27) entfernt alle Zeichen bis zum nächsten Zeilenende aus `cin`. Das ist ***unbedingt notwendig***, weil sonst das falsche Zeichen vorn in `cin`, welches den Fehler ausgelöst hat, ***nie*** entfernt würde und immer wieder den gleichen Fehler auslösen würde.

11.4 Zahlen beim Ausgeben formatieren (mit Strom-Methoden und Manipulatoren)

Wenn man Zahlen ausgibt (Ganz- oder Bruchzahlen), will man sie häufig ***formatieren***. Z. B. will man sie, unabhängig von ihrer Größe, in einer bestimmten ***Breite*** ausgeben, um sie so auf dem Bildschirm in übersichtlichen Spalten anzuordnen. Oder man will Bruchzahlen ***mit Exponent*** (z. B. `12.345e+2`) bzw. ***ohne Exponent*** (z. B. `1234.5`) ausgeben und die Anzahl der ***Nachpunktstellen*** (precision) festlegen. Das folgende Beispiel zeigt die wichtigsten Möglichkeiten, Zahlen beim Ausgeben in einen Strom zu formatieren:

```
1   // Datei EinAus14.cpp
2   /* ------------------------------------------------------------------------
3   Daten werden mit "<<" in einem Strom ausgegeben und dabei formatiert.
4   Anstelle von cout koennte auch ein anderer Ausgabestrom verwendet werden.
5   ------------------------------------------------------------------------ */
6   #include <iostream>
7   #include <iomanip>  // fuer die Manipulatoren setw, setfill, ...
8   using namespace std;
9   // ------------------------------------------------------------------------
10  // Ganzzahlen formatieren mit Elementmethoden des Stroms:
11  void f01(ostream & os) {
12     os.width(7);                                // Mindestlaenge
13     os.fill('-');                               // Fuellzeichen
14     os.setf(ios::left, ios::adjustfield);       // left, right, internal
15     os.setf(ios::oct,  ios::basefield);         // oct, dec, hex, nichts
16  } // f01
17  // ------------------------------------------------------------------------
18  // Ganzzahlen formatieren mit Manipulatoren:
19  void m01(ostream & os) {
20     os << setw(7)                               // Mindestlaenge
21        << setfill('-')                          // Fuellzeichen
22        << resetiosflags(ios::adjustfield)
23        <<   setiosflags(ios::left)              // left, right, internal
24        << resetiosflags(ios::basefield)
25        <<   setiosflags(ios::oct)               // oct, dec, hex, nichts
26     ;
27  } // m01;
28  // ------------------------------------------------------------------------
29  // Bruchzahlen formatieren mit Elementmethoden des Stroms:
30  void f02(ostream & os) {
31     os.width(17);                               // Mindestlaenge
32     os.fill('.');                               // Fuellzeichen
33     os.setf(ios::right,      ios::adjustfield); // left, right, internal
34     os.setf(ios::showpos  |  ios::uppercase);   // noshowpos, nouppercase
35     os.setf(ios::scientific, ios::floatfield);  // fixed, scientific, nichts
36     os.precision(3);                            // Anz. Nachpunktstellen
37  } // f02
38  // ------------------------------------------------------------------------
39  // Bruchzahlen formatieren mit Manipulatoren:
40  void m02(ostream & os) {
41     os << setw(17)                              // Mindestlaenge
42        << setfill('.')                          // Fuellzeichen
```

```
43        << resetiosflags(ios::adjustfield)
44        <<   setiosflags(ios::right)            // left, right, internal
45        <<   setiosflags(ios::showpos)          // noshowpos   ('+' oder nix)
46        <<   setiosflags(ios::uppercase)        // nouppercase ('E' oder 'e')
47        << resetiosflags(ios::floatfield)
48        <<   setiosflags(ios::scientific)       // fixed, scientific, nichts
49     ;
50  } // m02
51  // ------------------------------------------------------------------------
52  int main() {
53     cout << "EinAus14: Jetzt geht es los!"          << endl;
54     // Formatierungen beziehen sich nur auf die naechste Ausgabe (hier: 63),
55     // nicht auf spaetere Ausgaben (hier: 27):
56     f01(cout); cout << 63 << 27                     << endl;
57
58     // Mehrere Ausgaben (hier: 63 und 27) formatieren (gut, dass die
59     // Formatierungsbefehle in einer Prozedur, f01, stehen):
60     f01(cout); cout << 63;   f01(cout); cout << 27 << endl;
61
62     // Gleiche Formatierung mit den Manipulatoren in der Prozedur m01:
63     m01(cout); cout << 63 << 27 << endl;
64
65     // Gleiche Formatierung direkt mit Manipulatoren (ohne Prozedur):
66     cout << setw(7)
67          << setfill('-')
68          << resetiosflags(ios::adjustfield)
69          <<   setiosflags(ios::left)
70          << resetiosflags(ios::basefield)
71          <<   setiosflags(ios::oct)
72          << 63 << 27                               << endl;
73
74     // Bruchzahlen formatieren (mit Elementfunktionen bzw. Manipulatoren):
75     f02(cout); cout << 123.456789                  << endl;
76     m02(cout); cout << 123.456789                  << endl;
77
78     cout << "EinAus14: Das war's erstmal!"          << endl;
79  } // main
80  /* ------------------------------------------------------------------------
81  Ausgabe des Programms EinAus14:
82
83  EinAus14: Jetzt geht es los!
84  77-----33
85  77-----33-----
86  77-----33
87  77-----33
88  .......+1.235E+02
89  .......+1.235E+02
90  EinAus14: Das war's erstmal!
91  ----------------------------------------------------------------------- */
```

Die hier gezeigten Formatierungsbefehle haben nur bei der Ausgabe von ***Zahlen*** (Ganzzahlen bzw. Bruchzahlen) eine Wirkung. Sie haben ***keine*** Auswirkung auf die Ausgabe von ***Strings***.

Wenn man eine ***Ausgabebreite*** festlegt (mit der Methode `width` oder mit dem Manipulator `setw`) werden dadurch auf keinen Fall ***Stellen der Zahl abgeschnitten***. Auch wenn die Zahl "zu lang" ist, wird sie ***vollständig*** ausgegeben ("Besser das Layout ist gestört als dass die Tausenderziffer fehlt"). Gleitpunktzahlen werden beim Ausgeben ***gerundet*** (siehe Zeile 74 und 87).

Man beachte: `ios::showpos` und `ios::uppercase` sind Bits (flags), die man ***einzeln*** setzen kann. Im Gegensatz dazu bilden die Bits `ios::right`, `ios::left` und `ios::internal` eine

Gruppe namens `ios::adjustfield`. Innerhalb einer Gruppe "gilt" nur das (von links nach rechts gesehen) ***erste*** 1-Bit (`true`-Bit), die folgenden Bits sind wirkungslos. Deshalb werden durch den ***Methodenaufruf***

```
14 os.setf(ios::left, ios::adjustfield);
```

alle Bits der Gruppe `ios::adjustfield` auf `0` und dann (nur) das Bit `ios::left` auf `1` gesetzt. Wenn man ***Manipulatoren*** benützt, erreicht man den gleichen Effekt so:

```
20 os << ...
22    << resetiosflags(ios::adjusfield) // Alle Bits der Gruppe auf 0
23    <<   setiosflags(ios::left)       // Das left-Bit auf 1
24    << ...
26 ;
```

Entsprechendes gilt auch für die Bit-Gruppen `ios::basefield` und `ios::floatfield`.

Mit den hier skizzierten Möglichkeiten, Zahlen bei der Ausgabe zu formatieren, wird man am besten durch ***Experimente am Rechner*** vertraut. Modifizieren Sie das Programm `EinAus14` und beobachten Sie die Auswirkungen auf die Ausgaben. In den Kommentaren sind an einigen Stellen sinnvolle Modifikationen angedeutet (statt `left` kann man auch `right` oder `internal` angeben etc.). Noch besser: Schreiben Sie ein Programm, welches eine Zahl und die "Formatierungswünsche des Benutzers" einliest und die Zahl entsprechend formatiert wieder ausgibt.

11.5 Zahlen formatieren ohne sie auszugeben

Bei der ***Ausgabe*** in einen Strom werden die Daten im allgemeinen von einer ***internen*** ("binären") ***Darstellung*** in eine ***externe, formatierte Darstellung als String*** umgewandelt. Manchmal möchte man diese ***Umwandlung*** durchführen lassen, ***ohne*** die Daten tatsächlich ***auszugeben***.

Beim ***Einlesen*** aus einem Strom werden die Daten im allgemeinen von einer ***externen, formatierten Darstellung als String*** in eine ***interne*** ("binäre") ***Darstellung*** umgewandelt. Manchmal hat man gewisse Daten (vor allem Zahlen) schon als ***String*** eingelesen und möchte sie in ihre ***interne Darstellung umwandeln*** lassen (natürlich ohne sie noch mal ***einlesen*** zu müssen).

Solche "***Umwandlungen ohne Ein-/Ausgabe***" kann mit mit Hilfe spezieller Strömen durchführen, die einen ***String*** als ***Datensenke*** bzw. ***Datenquelle*** verwenden. In gewisser Weise ist es also möglich, Daten ***zu einem String "auszugeben"*** bzw. ***aus einem String "einzulesen"***.

Der C++-Standard sieht zwei ***Versionen*** von ***String-Strömen*** vor: Die ältere Version basiert auf den alten ***C-Strings*** und wird in der Kopfdatei `<strstream.h>` deklariert. Die neuere Version basiert auf ***C++-Strings*** und wird in der Kopfdatei `<sstream>` deklariert.

Das folgende Beispielprogramm zeigt, wie die modernen C++-String-Ströme im Prinzip funktionieren. Besonders interessant werden sie, wenn man sie mit den ***Formatierungsbefehlen*** kombiniert, die im vorigen Abschnitt behandelt wurden.

```
1   // Datei EinAus16.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert einen Ausgabestrom, der in einen C++-String ausgibt.
4   Der Gnu-Cygnus-Compiler, Version 2.95.2 kennt leider nur die aelteren
5   String-Stroeme in <strstream.h>, nicht die modernen in <sstream>.
6   ------------------------------------------------------------------------ */
7   #include <sstream>  // fuer Klassen ostringstream und string
8   #include <iostream>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  int main() {
12     cout << "EinAus16: Jetzt geht es los!" << endl;
13     ostringstream oss;
14
15     oss  << "Betrag: " << 25.55; // Daten in den Strom oss ausgeben
16     cout << oss.str()  << endl;  // Den C-String von oss nach cout ausgeben
17
18     oss  << " DM.";              // Weitere Daten nach oss ausgeben
19     cout << oss.str()  << endl;  // Den C-String von oss nach cout ausgeben
20
21   //oss.seekp(0);                // An den Anfang von oss zurueckgehen
22                                  // Verfaelscht oss.str().size() ??
23
24     oss.str("");                 // Besser: Den String von oss loeschen
25     oss  << "Amount: " << 12.34 << " Euro.";
26     cout << oss.str()  << endl;  // Den C-String von oss nach cout ausgeben
27
28     int len = oss.str().size();
29     cout << "Momentane Anzahl Zeichen im Strom oss: " << len << endl;
30
31     cout << "EinAus16: Das war's erstmal!" << endl;
32  } // main
33  /* ------------------------------------------------------------------------
34  Ausgabe des Programms EinAus16:
35
36  EinAus16: Jetzt geht es los!
37  Betrag: 25.55
38  Betrag: 25.55 DM.
39  Amount: 12.34 Euro.
40  Momentane Anzahl Zeichen im Strom oss: 19
41  EinAus16: Das war's erstmal!
42  ------------------------------------------------------------------------ */
```

Das Beispielprogramm `EinAus15` (hier nicht wiedergegeben) demonstriert ganz entsprechend die *älteren* C-String-Ströme, die auf *C-Strings* basieren.

12 Klassen als Module und Baupläne für Module

Das Konzept einer ***Klasse*** wird hier als eine ***Weiterentwicklung*** des ***Modulkonzepts*** dargestellt. Die abstrakten, ***sprachunabhängigen Definitionen*** der Begriffe ***Modul***, ***Klasse*** und ***Objekt*** werden durch eine kleine C++-Klasse illustriert.

Ein ***Modul*** ist ein ***Behälter*** für ***Unterprogramme***, ***Variablen*** , ***Konstanten***, ***Typen*** und ***Module*** etc., der aus einem ***sichtbaren*** und einem ***unsichtbaren*** Teil besteht. Nur auf die Größen im ***sichtbaren*** Teil des Moduls kann man ***von außen direkt zugreifen***, die Größen im ***unsichtbaren*** Teil sind vor direkten Zugriffen ***geschützt***.

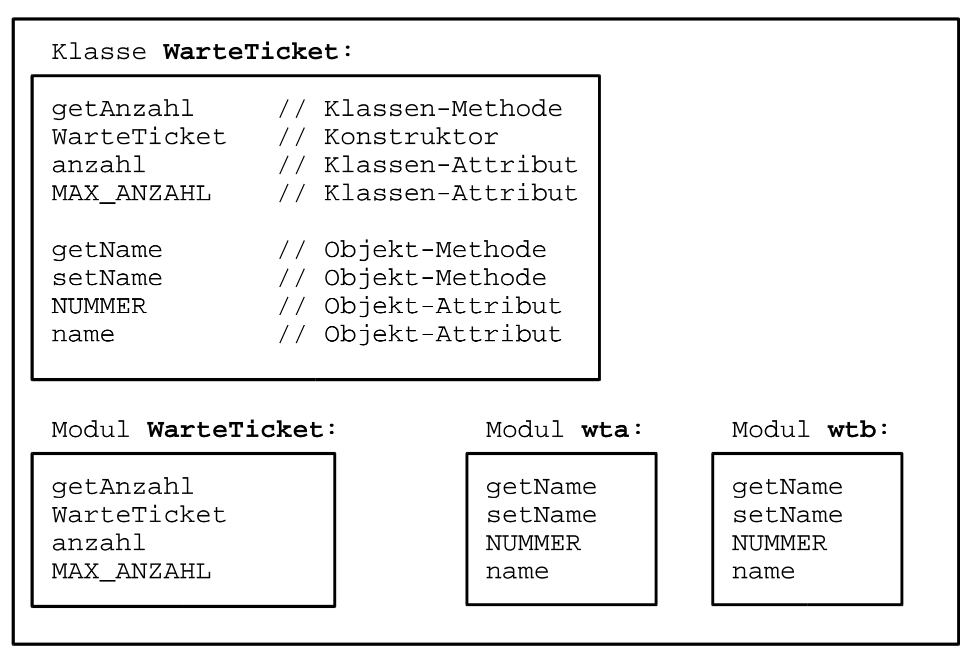
Eine ***Klasse*** ist gleichzeitig ein ***Modul*** und ein ***Bauplan für Module***. Module, die nach einem solchen Bauplan gebaut wurden, werden meist als ***Objekte*** (der Klasse) bezeichnet. Eine Klasse enthält ***Elemente***. Durch spezielle Notationen wird von jedem Element festgelegt, ob es zum ***Modulaspekt*** oder zum ***Bauplanaspekt*** der Klasse gehört.

Die Klasse `WarteTicket` im C++-Programm `Klassen10` beruht auf folgender Idee: Am Eingang einiger Wartezimmer steht ein Automat, an dem man sich ein "Warteticket" ziehen kann. Auf dem Ticket steht eine ***Nummer*** die entscheidet, wann man "dran kommt". Ein Objekt der Klasse `WarteTicket` enthält zusätzlich zu der (unveränderbaren) ***Nummer*** auch noch den ***Namen*** der wartenden Person. Dieser Name ist ***veränderbar***, d. h. die wartende Person darf ihr Ticket an eine andere Person weitergeben.

```
1   // Datei Klassen10.cpp
2   /* ------------------------------------------------------------------------
3   Definition einer einfachen Klasse namens WarteTicket. Jedes Objekt dieser
4   Klasse enthaelt eine (unveraenderbare) fortlaufende Nummer und einen (ver
5   aenderbaren) Namen. Die Klasse selbst enthaelt ein Attribut anzahl und
6   zaehlt darin, wieviele Objekte schon erzeugt wurden. Die Nummern laufen
7   von 1 bis MAX_ANZAHL und beginnen dann wieder bei 1.
8   ------------------------------------------------------------------------ */
9   #include <string>
10  #include <iostream>
11  using namespace std;
12  // ------------------------------------------------------------------------
13  // Definition der Klasse WarteTicket:
14  class WarteTicket {
15
16     // Elementgruppe 1: Aussen       sichtbare Klassenelemente (2 Elemente):
17     public:
18        static int getAnzahl() {return anzahl;}
19        WarteTicket (string name) : NUMMER(++anzahl), name(name) {
20           if (anzahl == MAX_ANZAHL) anzahl = 0;
21        }
22
23     // Elementgruppe 2: Aussen nicht sichtbare Klassenelemente (2 Elemente):
24     private:
25        static int       anzahl;           // Dies ist nur eine Deklaration!
26        static int const MAX_ANZAHL = 100; // Dies ist     eine Definition.
27
28     // Elementgruppe 3: Aussen       sichtbare  Objektelemente (3 Elemente):
29     public:
30        string getName ()            {return name;}
31        void   setName (string name) {this->name = name;}
32        int const NUMMER;                  // Definition! (Wert muss vom
33                                           // Konstruktor festgelegt werden)
34
35     // Elementgruppe 4: Aussen nicht sichtbare  Objektelemente (1 Element):
36     private:
```

```
37        string name;                     // Dies ist eine Definition.
38  }; // class WarteTicket
39  // ------------------------------------------------------------------------
40  // Definition des Klassenattributs anzahl:
41  int WarteTicket::anzahl = 0;
42  // ------------------------------------------------------------------------
43  int main() {
44     cout << "Klassen10: Jetzt geht es los!" << endl;
45
46     WarteTicket wta("Otto Meyer");
47     WarteTicket wtb("Sabine Mueller");
48
49     cout << "wta, Nr: " << wta.NUMMER << ", Name: " << wta.getName() << endl;
50     cout << "wtb, Nr: " << wtb.NUMMER << ", Name: " << wtb.getName() << endl;
51     cout << "WarteTicket::getAnzahl(): " << WarteTicket::getAnzahl() << endl;
52
53     wta.setName("Carola Schmidt");
54
55     cout << "wta, Nr: " << wta.NUMMER << ", Name: " << wta.getName() << endl;
56     cout << "Klassen10: Das war's erstmal!" << endl;
57  } // main
58  /* ------------------------------------------------------------------------
59  Ausgabe des Programms Klassen10:
60
61  Klassen10: Jetzt geht es los!
62  wta, Nr: 1, Name: Otto Meyer
63  wtb, Nr: 2, Name: Sabine Mueller
64  WarteTicket::getAnzahl: 2
65  wta, Nr: 1, Name: Carola Schmidt
66  Klassen10: Das war's erstmal!
67  ------------------------------------------------------------------------ */
```

Wenn der Ausführer das Programm `Klasse10` bis zur Zeile `48` ausgeführt hat, existieren ***3 Module***: Der Modul `WarteTicket` und die beiden Module (Objekte) `wta` und `wtb`, die "nach dem Bauplan `WarteTicket` gebaut wurden". Das folgende Diagramm soll diese wichtige Grundtatsache veranschaulichen:



Die Klasse `WarteTicket` enthält ***acht Elemente***. Die ersten ***vier Elemente*** (`getAnzahl`, `WarteTicket` `anzahl` und `MAX_ANZAHL`) gehören zum ***Modulaspekt*** der Klasse und somit zum Modul `WarteTicket`. Die restlichen ***vier Elemente*** (`getName`, `setName`, `NUMMER` und `name`) gehören zum ***Bauplanaspekt*** der Klasse und werden somit in jedes ***Objekt*** eingebaut, welches nach dem Bauplan `WarteTicket` erzeugt wird.

Im Diagramm erkennt man (hoffentlich) leicht, dass die beiden Module `wta` und `wtb` sich etwa so ähnlich sind wie zwei Häuser, die nach dem gleichen Bauplan gebaut wurden.

Die Elemente im Modul `WarteTicket` kann man mit den folgenden Namen bezeichnen:

`WarteTicket::getAnzahl`, `WarteTicket::WarteTicket`, `WarteTicket::anzahl` und `WarteTicket::MAX_ANZAHL` (siehe Zeile 41 und 51). Allerdings sind die beiden Elemente `WarteTicket::anzahl` und `WarteTicket::MAX_ANZAHL` außerhalb des Moduls `WarteTicket` ***nicht sichtbar*** (privat) und den ***Konstruktor*** kann man auch außerhalb des Moduls WarteTicket meist mit seinem einfachen Namen `WarteTicket` bezeichnet statt mit seinem vollen Namen `WarteTicket::WarteTicket`.

Die Elemente im Modul `wta` kann man mit den Namen `wta.getName`, `wta.setName`, `wta.NUMMER` und `wta.name` bezeichnen (siehe z. B. Zeile 44 und 48). Allerdings ist das Element `wta.name` außerhalb des Moduls `wta` ***nicht sichtbar*** (privat).

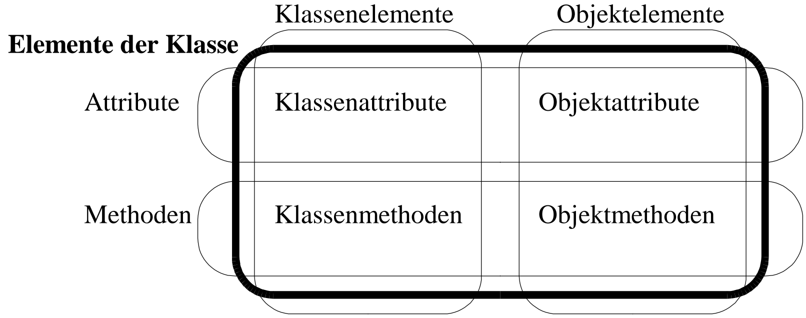
Die Begriffe "***Modulaspekt***" und "***Bauplanaspekt***" sind (noch?) ***nicht*** sehr verbreitet. Hier eine ***Übersetzung*** in die offiziellen und verbreiteten Begriffe:

Hier eingeführte Bezeichnung:	**Offizielle Bezeichnung:**
Elemente der Klasse, die zum ***Modulaspekt*** gehören:	***Klassenelemente***
Elemente der Klasse, die zum ***Bauplanaspekt*** gehören:	***Objektelemente***

In C++ (und in Java) gehören die ***Konstruktoren*** einer Klasse (soweit vorhanden), der ***Destruktor*** (falls vorhanden) und alle mit ***static*** gekennzeichneten Elemente zum ***Modulaspekt***, sind also ***Klassen-Elemente***. Alle ***übrigen*** (***nicht*** mit ***static*** gekennzeichneten) Elemente gehören zum ***Bauplanaspekt*** und sind somit ***Objektelemente***.

Achtung: Die offizielle Bezeichnung ***Klassenelement*** verstößt gegen eine wichtige "Regel der allgemeinen Sprachlogik". Üblicherweise ist jedes ***Mitglied eines Vereins*** ein ***Vereinsmitglied*** und jedes ***Element einer Menge*** ist ein ***Mengenelement*** etc. Für Klassen gilt dagegen: Nicht jedes ***Element einer Klasse*** ist auch ein ***Klassenelement***, da einige Elemente ***Objektelemente*** sind.

Jedes Element einer Klasse ist entweder eine ***Methode*** (ein Unterprogramm) oder ein ***Attribut*** (eine Variable oder Konstante, engl. a field). Diese Unterscheidung ist ***unabhängig*** von der Unterscheidung zwischen ***Klassenelement***en und Objektelementen, wie das folgende Diagramm deutlich machen soll:



In ***C++*** gehören die ***Konstruktoren*** und der ***Destruktor*** einer Klasse zu den ***Klassenelementen*** (in ***Java*** gibt es keine Destruktoren und die Konstruktoren gehören ***nicht*** zu den Methoden, sondern bilden eine ***eigene*** Gruppe von Elementen).

12.1 Deklarationen und Definitionen von und in Klassen

In C++ kann man eine Klasse (ähnlich wie andere Größen auch) ***deklarieren*** und ***definieren***. Nachdem man eine Klasse nur ***deklariert*** hat (d. h. nachdem man dem Ausführer versprochen hat, sie in irgendeiner Quelldatei des Programms zu definieren), kann man sie schon ein bisschen benutzen, aber nur ***sehr stark eingeschränkt***, etwa so:

```
1       class otto;                       // Deklaration einer Klasse namens otto
2       otto * p1;                        // Adressvar. vereinbaren  erlaubt
3       typedef otto * AdresseVonOtto;    // Alias-Namen deklarieren erlaubt
4       otto objekt1;                     // Objekt vereinbaren      verboten!
5       otto * p2 = new otto();           // Objekt erzeugen         verboten!
```

In jeder Datei, in der man eine Klasse "richtig" benutzen will (z. B. zum Vereinbaren von Objekten), muss man sie ***definieren***. Zur Erinnerung: Jede Klasse ist ein ***Typ*** und für Typen ist die ***ODR*** (one definition rule) ***nicht gültig***!

Innerhalb einer ***Klassendefinition*** muss man jedes Element der Klasse ***vereinbaren*** (d. h. deklarieren oder definieren). Das kann im Prinzip nach einer der folgenden beiden Vorgehensweisen geschehen:

Vorgehensweise 1: ***Innerhalb*** der Klassendefinition ***deklariert*** man das Element nur und ***definiert*** es dann ***außerhalb*** der Klassendefinition.

Vorgehensweise 2: Mann ***definiert*** das Element gleich ***innerhalb*** der ***Klassendefinition***.

Allerdings hat man nicht bei allen Elementen die ***Wahl*** zwischen beiden Vorgehensweisen (sonst könnte ja jeder in C++ programmieren :-). Hier die genauen Regeln:

K-DekDef1: Für ***Objektattribute*** muss man die ***Vorgehensweise2*** anwenden.

K-DekDef2: Für ***Klassenattribute*** muss man die ***Vorgehensweise1*** anwenden. ***Ausnahme***: Für ***const-int***-Klassenattribute darf man ***wahlweise*** Vorgehensweise1 oder 2 anwenden.

K-DekDef3: Für ***Methoden*** (Objektmethoden und Klassenmethoden, inklusive Konstruktoren und Destruktor) darf man ***wahlweise*** Vorgehensweise1 oder 2 anwenden.

Als erstes Beispiel zur Illustration dieser Regeln sollte man sich die Klasse `WarteTicket` (im Programm `Klassen10` im vorigen Abschnitt) noch einmal ansehen. Dort wurden möglichst viele Elemente schon innerhalb der Klassendefinition vollständig definiert. Nur ein Klassenattribut (`anzahl`) musste nach Vorgehensweise1 behandelt und außerhalb definiert werden.

Als zweites Beispiel folgt hier eine andere, praxisnäher strukturierte Version der gleichen Klasse. Bei dieser Version wird die Klasse in einer Kopfdatei `WarteTicket.h` definiert und die zugehörigen Element-Definitionen stehen in einer Implementierungsdatei `WarteTicket.cpp`. Ein kleines Testprogramm in der Datei Warte `WarteTicket_Tst.cpp` inkludiert die Kopfdatei und benutzt die Klasse.

```
1   // Datei WarteTicket.h
2   /* ------------------------------------------------------------------------
3   Definition einer einfachen Klasse namens WarteTicket. Jedes Objekt dieser
4   Klasse enthaelt eine (unveraenderbare) fortlaufende Nummer und einen (ver
5   aenderbaren) Namen. Die Klasse selbst enthaelt ein Attribut anzahl und
6   zaehlt darin, wieviele Objekte schon erzeugt wurden. Die Nummern laufen
7   von 1 bis MAX_ANZAHL und beginnen dann wieder bei 1.
8   ------------------------------------------------------------------------ */
9   // Da diese Kopfdatei eine Definition (der Klasse WarteTicket) enthaelt,
10  // muss mehrfaches Inkludieren verhindert werden:
11  #ifndef WarteTicket_h
12  #define WarteTicket_h
13
14  #include <string>
15  using namespace std;
16  // ------------------------------------------------------------------------
17  // Definition der Klasse WarteTicket:
18  class WarteTicket {
19
20     // Elementgruppe 1: Aussen       sichtbare Klassenelemente (2 Elemente):
21     public:
22        static int getAnzahl();       // Dies ist eine Deklaration
23        WarteTicket (string name);    // Dies ist eine Deklaration
24
25     // Elementgruppe 2: Aussen nicht sichtbare Klassenelemente (2 Elemente):
26     private:
27        static int       anzahl;      // Dies ist eine Deklaration!
28        static int const MAX_ANZAHL;  // Dies ist eine Deklaration!
29
30     // Elementgruppe 3: Aussen       sichtbare  Objektelemente (3 Elemente):
31     public:
32        string getName ();            // Dies ist eine Deklaration
33        void   setName (string name); // Dies ist eine Deklaration
34        int const NUMMER;             // Dies ist eine Definition
35
36     // Elementgruppe 4: Aussen nicht sichtbare  Objektelemente (1 Element):
37     private:
38        string name;                  // Dies ist eine Definition.
39  }; // class WarteTicket
40  /* ------------------------------------------------------------------------
41  In dieser Klassendefinition wurden moeglichst viele Elemente nur deklariert.
42  Die Definitionen dieser Elemente stehen in der Implementierungsdatei
43  WarteTicket.cpp. Nur die Objekt-Attribute (NUMMER und name) mussten schon
44  hier innerhalb der Klassendefinition definiert werden.
45  ------------------------------------------------------------------------ */
46
47  #endif // ifndef WarteTicket_h
```

```
48  // Datei WarteTicket.cpp
49  /* ------------------------------------------------------------------------
50  Definitionen aller Elemente der Klasse WarteTicket, die nicht schon
51  innerhalb der Klassen-Definition (in der Kopfdatei WarteTicket.h)
52  definiert werden mussten.
53  ------------------------------------------------------------------------ */
54  #include "WarteTicket.h"
55  #include <string>
56
57  // Elementgruppe 1: Aussen       sichtbare Klassenelemente (2 Elemente):
58  int WarteTicket::getAnzahl() {return anzahl;}
59  WarteTicket::WarteTicket (string name) : NUMMER(++anzahl), name(name) {
60     if (anzahl == MAX_ANZAHL) anzahl = 0;
61  }
62
63  // Elementgruppe 2: Aussen nicht sichtbare Klassenelemente (2 Elemente):
64  int       WarteTicket::anzahl;
65  int const WarteTicket::MAX_ANZAHL = 100;
66
67  // Elementgruppe 3: Aussen       sichtbare  Objektelemente (3 Elemente):
68  string WarteTicket::getName ()            {return name;}
69  void   WarteTicket::setName (string name) {this->name = name;}
70  // Das Objektattribut NUMMER ist schon definiert.
71
72  // Elementgruppe 4: Aussen nicht sichtbare  Objektelemente (1 Element):
73  // Das Objektattribut name ist schon definiert.
74  // ------------------------------------------------------------------------
```

```
1   // Datei WarteTicket_Tst.cpp
2   /* ------------------------------------------------------------------------
3   Ein kleines Programm zum Testen der Klasse WarteTicket.
4   ------------------------------------------------------------------------ */
5   #include "WarteTicket.h"
6   #include <iostream>
7   using namespace std;
8   // ------------------------------------------------------------------------
9   int main() {
10     cout << "WarteTicket_Tst: Jetzt geht es los!" << endl;
11
12     WarteTicket wta("Otto Meyer");
13     WarteTicket wtb("Sabine Mueller");
14
15     cout << "wta, Nr: " << wta.NUMMER << ", Name: " << wta.getName() << endl;
16     cout << "wtb, Nr: " << wtb.NUMMER << ", Name: " << wtb.getName() << endl;
17     cout << "WarteTicket::getAnzahl(): " << WarteTicket::getAnzahl() << endl;
18
19     wta.setName("Carola Schmidt");
20
21     cout << "wta, Nr: " << wta.NUMMER << ", Name: " << wta.getName() << endl;
22     cout << "WarteTicket_Tst: Das war's erstmal!" << endl;
23  } // main
24  /* ------------------------------------------------------------------------
25  Ausgabe des Programms WarteTicket_Tst:
26
27  WarteTicket_Tst: Jetzt geht es los!
28  wta, Nr: 1, Name: Otto Meyer
29  wtb, Nr: 2, Name: Sabine Mueller
30  WarteTicket::getAnzahl(): 2
31  wta, Nr: 1, Name: Carola Schmidt
32  WarteTicket_Tst: Das war's erstmal!
33  ------------------------------------------------------------------------ */
```

12.2 Sichtbarkeitsregeln innerhalb einer Klasse

In C++ sind die ***Sichtbarkeitsregeln*** innerhalb einer ***Klasse*** ganz anders als innerhalb einer ***Datei*** oder innerhalb eines Unterprogramms. Innerhalb einer ***Datei*** (oder innerhalb eines Unterprogramms) ist die ***Reihenfolge*** der einzelnen Vereinbarungen ***wichtig***, wie das folgende Beispiel deutlich machen soll:

```
1   // Datei B17.cpp
2   ...
3   int otto = 17;
4   void put() {
5      cout << otto << endl; // Erlaubt,  otto ist hier            sichtbar
6      cout << emil << endl; // Verboten, emil ist hier noch nicht sichtbar
7   } // put
8   int emil = 25;
9   ...
```

Eine (direkt) in einer ***Datei*** vereinbarte Größe ist erst ab ***der*** Stelle im Programmtext sichtbar, an der sie ***vereinbart*** (mindestens ***deklariert*** oder aber ***definiert***) wird.

Innerhalb einer ***Klasse*** spielt die Reihenfolge, in der die einzelnen Elemente vereinbart werden, (für den Compiler) ***keine Rolle***. Außerdem gelten folgende Zugriffsregeln:

Zugriffsregel 1: Innerhalb einer ***Objektmethode*** darf man alle ***Klassenmethoden*** aufrufen (auch ***private*** Klassenmethoden) und auf alle ***Klassenattribute*** zugreifen (auch auf private Klassenattribute). Innerhalb einer ***Klassenmethode*** darf man keine ***Objektmethode*** aufrufen und auf kein ***Objekt-Attribut*** zugreifen.
Zugriffsregel 2: Wenn man ein ***Objektattribut*** definiert (dies muss ***innerhalb*** der Klassendefinition geschehen, siehe oben), dann darf man das Attribut (die Variable bzw. Konstante) ***nicht*** initialisieren. Konstanten müssen und Variablen können ***per Konstruktor*** initialisiert werden.

Zugriffsregel 3: Wenn man ein ***Klassenattribut*** definiert (dies muss meistens ***außerhalb*** der Klassendefinition geschehen, siehe oben), darf man es initialisieren, dabei aber keine ***Objektfunktionen*** aufrufen oder auf ***Objektattribute*** zugreifen.

Die Regeln 1 und 3 gelten ganz entsprechend auch in Java und anderen objektorientierten Sprachen. Sie ergeben sich ***zwangsläufig*** aus der Tatsache, dass eine ***Klasse*** und alle ***Klassenelemente*** schon fertig erzeugt sein müssen, ***bevor*** das erste ***Objekt*** dieser Klasse ("entsprechend dem Bauplanaspekt der Klasse") erzeugt werden kann. Deshalb darf die Erzeugung eines ***Klassenelements*** nicht von irgendwelchen ***Objektelementen*** abhängig gemacht werden.

Alle Attribute einer Klasse, die der Programmierer ***nicht initialisiert***, werden ***automatisch*** vom Ausführer ***initialisiert*** (`int`-Attribute mit `0`, `float`-Attribute mit `0.0`, Attribute eines ***Adresstyps*** mit `NanA` alias `NULL` etc.).

Die ***privaten*** Elemente eines Moduls (einer Klasse oder eines Objekts) sind gegen ***direkte Zugriffe*** von ***außerhalb*** des Moduls geschützt. Hier ein Beispiel für einen ***indirekten Zugriff***: Eine öffentliche (public) Methode `m` darf auch auf ein privates Attribut `a` ihrer Heimatklasse zugreifen. Wenn man `m` aufruft, greift man damit indirekt auf das Attribut `a` zu. Da die Methode `m` öffentlich ist, darf man sie auch von Stellen außerhalb ihrer Heimatklasse aufrufen.

Auf ***Klassenelement***e greift man von Stellen außerhalb der Klasse mit Hilfe des ***Gültigkeitsbereichs-operators*** (scope operator) `"::"` zu, etwa so: `WarteTicket::getAnzahl()`. Auf ***Objektelement***e greift man von Stellen außerhalb des Objekts mit Hilfe der ***Punktnotation*** zu, etwa so: `wta.getName()`, `wtb.NUMMER` etc. Um die Leser eines Programms zu verwirren, kann man aber auch über einen ***Objekt-Namen*** mit der ***Punktnotation*** auf ***Klassenelemente*** zugreifen, etwa so: `wta.getAnzahl()`.

12.3 Eine Klasse Text "mit allem drum und dran"

Im folgenden Beispielprogramm wird eine Klasse namens `Text1` vereinbart. Jedes Objekt dieser Klasse besteht im wesentlichen aus einer Adresse eines `string`-Objekts. Das hat eine Reihe von ***Problemen*** zur Folge, wie man an der Ausgabe des Programms erkennen kann:

```
1   // Datei Klassen11.cpp
2   /* ------------------------------------------------------------------------
3   Eine Klasse namens Text1, noch "ohne alles drum und dran". Text1-Objekte
4   verhalten sich ganz anders, als eigentlich beabsichtigt.
5   ------------------------------------------------------------------------ */
6   #include <string>
7   #include <iostream>
8   using namespace std;
9   // ------------------------------------------------------------------------
10  // Definition der Klasse Text1:
11  class Text1 {
12     public:                                // Klassenelemente --------------
13        Text1(char * s);                    // Allgemeiner Konstruktor (K1)
14        Text1(int anzahl, char zeichen);    // Allgemeiner Konstruktor (K2)
15
16     public:                                // Objektelemente ---------------
17        string getString();                 // get-Methode
18        void   setString(string s);         // set-Methode
19     private:                               // Objektelemente ---------------
20        string * avs;                       // Attribut ("Adresse von string")
21  }; // class Text1
22  // ------------------------------------------------------------------------
23  // Definitionen der Methoden der Klasse Text1:
24
25  Text1::Text1(char * s) {                  // Allgemeiner Konstruktor (K1)
26     avs = new string(s);
27     cout << "K1:" << avs << endl;
28  } // Allgemeiner Konstruktor
29
30  Text1::Text1(int anzahl, char zeichen) {  // Allgemeiner Konstruktor (K2)
31     avs = new string(anzahl, zeichen);
32     cout << "K2:" << avs << endl;
33  } // Allgemeiner Konstruktor
34
35  string
36  Text1::getString() {                      // get-Methode
37     return *avs;
38  } // getString
39
40  void
41  Text1::setString(string s) {              // set-Methode
42     *avs = s;
43  } // setString
44  // ------------------------------------------------------------------------
45  // Eine speicherhungrige Prozedur:
46  void machWas() {
47     Text1 t1(10*1000*1000, '!'); // 10 MB gefuellt mit Ausrufezeichen
48     cout << '.' << flush;        // Ein Punkt '.' wird sofort ausgegeben
49  } // machWas
50  // ------------------------------------------------------------------------
51  int main() {
52     cout << "Klassen11: Jetzt geht es los!"      << endl;
53     Text1 t1("Hallo Susi!");
54     Text1 t2("Wie geht es?");
55     Text1 t3(t2);
56  // Text1 t4;
```

```
57
58  // cout << "(t1 == t2):     " << (t1 == t2)     << endl;
59
60     cout << "t1.getString(): " << t1.getString() << endl;
61     cout << "t2.getString(): " << t2.getString() << endl;
62     cout << "t3.getString(): " << t3.getString() << endl;
63
64     t2.setString("How are you?");
65     cout << "t2.getString(): " << t2.getString() << endl;
66     cout << "t3.getString(): " << t3.getString() << endl;
67
68     t1 = t2;
69     t1.setString("Hello!");
70
71     cout << "t1.getString(): " << t1.getString() << endl;
72     cout << "t2.getString(): " << t2.getString() << endl;
73
74     for (int i=0; i<10; i++) {machWas();}
75
76     cout << "Klassen11: Das war's erstmal!"      << endl;
77  } // main
78  /* ------------------------------------------------------------------------
79  Fehlermeldung des Gnu-Compilers, wenn Zeile 56 und 58 keine Kommentare sind:
80
81  Klassen11.cpp:56: no matching function for call to `Text1::Text1 ()'
82  Klassen11.cpp:25: candidates are: Text1::Text1(char *)
83  Klassen11.cpp:30:                 Text1::Text1(int, char)
84  Klassen11.cpp:21:                 Text1::Text1(const Text1 &)
85  Klassen11.cpp:58: no match for `Text1 & == Text1 &'
86  ------------------------------------------------------------------------
87  Ausgabe des Programms Klassen11:
88
89  Klassen11: Jetzt geht es los!
90  K1:0x45e0b28
91  K1:0x45e0b38
92  t1.getString(): Hallo Susi!
93  t2.getString(): Wie geht es?
94  t3.getString(): Wie geht es?
95  t2.getString(): How are you?
96  t3.getString(): How are you?
97  t1.getString(): Hello!
98  t2.getString(): Hello!
99  K2:0x45e0b48
100 .K2:0x45e0b58
101 .K2:0x65e19e8
102 .K2:0x75e1a10
103 .K2:0x85e1a38
104 .K2:0x95e1a60
105 .K2:0xa5e1a88
106 .out of memory
107 ---------------------------------------------------------------------- */
```

Problem 1: Warum wird die Ausführung des Programms `Klassen11` mit der Fehlermeldung `out of memory` abgebrochen? Am Anfang des Unterprogramms `machWas` wird zwar eine "große Variable" namens `t1` erzeugt, aber die wird doch am Ende des Unterprogramms wieder zerstört, oder?

Problem 2: In ***Zeile 64*** wird die Variable `t2` verändert (mit der Methode `setString`). Warum ist danach offensichtlich auch die Variable `t3` verändert (siehe Zeile 95 und 96)?

Problem 3: In ***Zeile 69*** wird die Variable `t1` verändert (mit der Methode `setString`). Warum ist danach offensichtlich auch die Variable `t2` verändert (siehe Zeile 97 und 98)?

Anmerkung: Die Probleme 2 und 3 ***ähneln*** sich, haben aber ***unterschiedliche*** Ursachen und Lösungen.

Problem 4: Schade, dass man ***Text***-Objekte nicht "einfach so" ***vereinbaren*** kann, ***ohne*** irgendwelche Parameter anzugeben (siehe Zeile 56 und 81 bis 84).

Problem 5: Schade, dass man ***Text***-Objekte nicht ***vergleichen*** kann (siehe Zeile 58 und 85).

Problem 6: Schade, dass man beim Ausgeben von Text-Objekten immer umständlich die Methode `getString`-aufrufen muss (siehe z. B. Zeile 60 bis 62).

Problem 7: Was bedeutet die ***Fehlermeldung des Compilers*** (Zeile 81 bis 84), insbesondere die letzte Zeile (***84***) genau? Warum kennt der Compiler ***drei*** Konstruktor-Kandidaten, wo doch in der Klasse ***Text*** nur ***zwei*** Konstruktoren vereinbart wurden?

Im folgenden Beispielprogramm (`Klassen12`) wird eine Klasse namens `Text2` vereinbart, diesmal aber "***mit allem drum und dran***". Damit sind folgende ***Elemente*** gemeint: Ein ***Standard-Konstruktor*** (d. h. ein Konstruktor ohne Parameter), ein ***Kopier-Konstruktor*** (d. h. ein Konstruktor mit einem Parameter vom Typ `Text const &`), ein ***Destruktor*** (immer ohne Parameter), ein ***Zuweisungsoperator***, ein ***Vergleichsoperator*** und ein (befreundeter) ***Ausgabeoperator***.

Indem man diese Elemente in der Klasse "richtig definiert" kann man die Probleme 1 bis 7 zum Verschwinden bringen.

```
1   // Datei Klassen12.cpp
2   /* ------------------------------------------------------------------------
3   Eine Klasse namens Text2, "mit allem drum und dran".
4   ---------------------------------------------------------------------- */
5   #include <string>
6   #include <iostream>
7   using namespace std;
8   // ------------------------------------------------------------------------
9   class Text2 {
10     public:                               // Klassenelemente --------------
11        Text2(char * s);                   // Allgemeiner Konstruktor (K1)
12        Text2(int anzahl, char zeichen);   // Allgemeiner Konstruktor (K2)
13        Text2();                           // Standard-Konstruktor    (K3)
14        Text2(Text2 const & t);            // Kopier-Konstruktor      (K4)
15        ~Text2();                          // Destruktor              (D)
16
17     public:                               // Objektelemente ---------------
18        string getString();                // get-Methode
19        void   setString(string s);        // set-Methode
20
21        Text2 const &                      // Zuweisungs-Operator =
22        operator =  (Text2 const & rs);
23
24        bool                               // Vergleichs-Operator ==
25        operator == (Text2 const & rs);
26
27     private:                              // Objektelemente ---------------
28        string * avs;                      // Attribut
29
30                                           // Freunde ----------------------
31     friend ostream &                      // Ausgabeoperator
32     operator << (ostream & os, Text2 const & t);
33  }; // class Text2
34  // ------------------------------------------------------------------------
35  // Definitionen der Methoden der Klasse Text2:
36  Text2::Text2(char * s) {                 // Allgemeiner Konstruktor (K1)
37     avs = new string(s);
38     cout << "K1:" << avs << endl;
```

```
39  } // Allgemeiner Konstruktor
40
41  Text2::Text2(int anzahl, char zeichen) { // Allgemeiner Konstruktor (K2)
42     avs = new string(anzahl, zeichen);
43     cout << "K2:" << avs;
44  } // Allgemeiner Konstruktor
45
46  Text2::Text2() {                        // Standar-Konstruktor     (K3)
47     avs = new string("");
48     cout << "K3:" << avs << endl;
49  } // Standard-Konstruktor
50
51  Text2::Text2(Text2 const & t) {         // Kopier-Konstruktor      (K4)
52     avs = new string(*t.avs);
53     cout << "K4:" << avs << endl;
54  } // Kopier-Konstruktor
55
56  Text2::~Text2() {                       // Destruktor              (D)
57     cout << "D:" << avs << endl;
58     delete avs;
59  } // Destruktor
60
61  string
62  Text2::getString() {                    // get-Methode
63     return *avs;
64  } // getString
65
66  void
67  Text2::setString(string s) {            // set-Methode
68     *avs = s;
69  } // setString
70
71  Text2 const &
72  Text2::operator = (Text2 const & rs) {  // Zuweisungs-Operator =
73     *this->avs = *rs.avs;
74     return rs;
75  } // operator =
76
77  bool
78  Text2::operator == (Text2 const & rs) { // Vergleichs-Operator ==
79     return *avs == *rs.avs;
80  } // operator ==
81
82  ostream &                               // Ausgabeoperator <<
83  operator << (ostream & os, Text2 const & t) {
84     os << *t.avs;
85     return os;
86  } // operator <<
87  // ------------------------------------------------------------------------
88  // Eine speicherhungrige Prozedur:
89  void machWas() {
90     Text2 t1(10*1000*1000, '!'); // 10 MB gefuellt mit Ausrufezeichen
91     cout << '.' << flush;        // Ein Punkt '.' wird sofort ausgegeben
92  } // machWas
93  // ------------------------------------------------------------------------
94  int main() {
95     cout << "Klassen12: Jetzt geht es los!" << endl;
96     Text2 t1("Hallo Susi!");
97     Text2 t2("Wie geht es?");
98     Text2 t3(t2);
99     Text2 t4;
100
```

```
101    cout << "(t1 == t2): " << (t1 == t2)    << endl;
102
103    cout << "t1: " << t1 << endl;
104    cout << "t2: " << t2 << endl;
105    cout << "t3: " << t3 << endl;
106    cout << "t4: " << t4 << endl;
107
108    t2.setString("How are you?");
109    cout << "t2: " << t2 << endl;
110    cout << "t3: " << t3 << endl;
111
112    t1 = t2;
113    t1.setString("Hello!");
114
115    cout << "t1: " << t1 << endl;
116    cout << "t2: " << t2 << endl;
117
118    for (int i=0; i<10; i++) {machWas();}
119
120    cout << "Klassen12: Das war's erstmal!" << endl;
121 } // main
122 /* ------------------------------------------------------------------------
123 Ausgabe des Programms Klassen12:
124
125 Klassen12: Jetzt geht es los!
126 K1:0x45e0b28
127 K1:0x45e0b38
128 K4:0x45e0b48
129 K3:0x45e0b58
130 (t1 == t2): 0
131 t1: Hallo Susi!
132 t2: Wie geht es?
133 t3: Wie geht es?
134 t4:
135 t2: How are you?
136 t3: Wie geht es?
137 t1: Hello!
138 t2: How are you?
139 K2:0x45e19b8.D:0x45e19b8
140 K2:0x45e19b8.D:0x45e19b8
141 K2:0x45e19b8.D:0x45e19b8
142 K2:0x45e19b8.D:0x45e19b8
143 K2:0x45e19b8.D:0x45e19b8
144 K2:0x45e19b8.D:0x45e19b8
145 K2:0x45e19b8.D:0x45e19b8
146 K2:0x45e19b8.D:0x45e19b8
147 K2:0x45e19b8.D:0x45e19b8
148 K2:0x45e19b8.D:0x45e19b8
149 Klassen12: Das war's erstmal!
150 D:0x45e0b58
151 D:0x45e0b48
152 D:0x45e0b38
153 D:0x45e0b28
154 --------------------------------------------------------------------- */
```

Problem 1 wird durch den ***Destruktor*** gelöst: Das Programm `Klassen12` wird ordnungsgemäß beendet, ohne die Fehlermeldung `out of memory`.

Problem 2 ist durch den ***Kopier-Konstruktor*** gelöst: In ***Zeile 108*** wird die Variable `t2` verändert (mit der Methode `setString`), die Variable `t3` bleibt dabei unverändert (siehe ***Zeile 135*** und ***136***).

Problem 3 ist durch den ***Zuweisungsoperator*** gelöst: In ***Zeile 113*** wird die Variable `t1` verändert (mit der Methode `setString`), die Variable `t2` bleibt dabei unverändert (siehe Zeile 137 und 138).

Problem 4 ist durch den ***Standard-Konstruktor*** gelöst: Man kann `Text2`-Objekte ***vereinbaren***, ***ohne*** irgendwelche Parameter anzugeben (siehe Zeile 99 und 134).

Problem 5 ist durch den ***Vergleichsoperator*** gelöst: Man kann `Text2`-Objekte miteinander ***vergleichen*** (siehe Zeile 101 und 130).

Problem 6 ist durch den ***Ausgabeoperator*** gelöst: Man kann `Text2`-Objekte jetzt direkt mit dem ***Operator*** << ausgeben, ohne die Funktion `getString` aufzurufen (siehe z. B. 103 bis 106 und 131 bis 134).

Problem 7, hier ist die ***Lösung***: Wenn der Programmierer keinen ***Kopierkonstruktor*** vereinbart, bekommt er einen vom Compiler "***geschenkt***". Der geschenkte Kopier-Konstruktor kopiert einfach "***elementweise***", was im Falle der Klasse `Text1` nicht die gewünschte Wirkung hat (siehe oben Problem 2).

12.4 Die Klasse Birnen, deren Objekte sich wie Zahlen verhalten

Computer sind heute zwar wesentlich schneller als vor 30 Jahren, aber das ***Rechnen mit Ganzzahlen*** funktioniert auch auf modernen Rechnern nicht besser als damals:

Schwächen moderner Hardware bei der Darstellung von ***Ganzzahlen***:

1. Es gibt ***keine*** Bitkombination, die einen ***undefinierten Ganzzahlwert*** darstellt (entsprechend den `NaN`-Werten bei Gleitpunktzahlen oder den `NanA`-Werten bei Adressen).

2. Werden negative Zahlen im ***Zweierkomplement*** dargestellt (eine immer noch verbreitete Unsitte), gibt es ***mehr negative*** Zahlen als ***positive***. Damit werden scheinbar harmlose Operationen wie das Vorzeichen Minus und die Betragsfunktion `abs` zu gefährlichen Befehlen, die schief gehen können (z. B. ist dann `n != -n` nicht immer gleich `true` und `abs(n)` ist nicht immer positiv!).

3. Es gibt keine Bitkombinationen für ***Plus-Unendlich*** und ***Minus-Unendlich*** (entsprechend den Werten `infinity` und `-infinity` bei Gleitpunktzahlen).

Hinzu kommt folgende ***Schwäche der Sprache C++***: ***Ganzzahlüberläufe*** lösen ***keine*** Ausnahme aus und liefern kein "offensichtlich falsches" Ergebnis.

Besonders bedauerlich ist, dass die doch recht neue Sprache ***Java*** diese Schwäche von C++ übernommen hat. Schon seit 1980 gibt es die Sprache ***Ada*** (heute auf allen wichtigen Plattformen implementiert), in der Ganzzahlüberläufe eine Ausnahme auslösen (es geht also auch besser).

Das waren die schlechten Nachrichten. Die gute Nachricht: ***C++*** bietet dem Programmierer die Möglichkeit, die hier skizzierten (Hard- und Software-) Schwächen beim Umgang mit Ganzzahlen (mit ein bisschen zusätzlicher Programmierarbeit) ***unwirksam*** zu machen. Die folgende Klasse `Birnen` zeigt wie:

```
1   // Datei Birnen.h
2   /* ------------------------------------------------------------------------
3   ...
4   --------------------------------------------------------------------- */
5   // Da diese Kopfdatei eine Defintion (und nicht nur Deklarationen) ent-
6   // haelt, muss mehrfaches Inkludieren verhindert werden:
7   #ifndef Birnen_h
8   #define Birnen_h
9   // ------------------------------------------------------------------------
10  #include <iostream>
11  using namespace std;
```

```
12
13 class Birnen {
14    // ------------------------------------------------------------------------
15    // Klassenelemente:
16    public:
17       typedef int ganz;                // Ein geeigneter Ganzzahltyp
18       static ganz const MIN = -500;    // Kleinster     Birnen-Wert
19       static ganz const MAX = +800;    // Groesster     Birnen-Wert
20       static ganz const ERR = MIN-1;   // Undefinierter Birnen-Wert
21
22       Birnen(ganz g = ERR);            // Standard- und allgem. Konstruktor
23    // ------------------------------------------------------------------------
24    // Objektelemente:
25    private:
26       ganz wert;                       // Der Wert dieses Birnen-Objekts
27    public:
28       // Operatoren zum Rechnen mit Birnen-Objekten:
29       Birnen operator + (Birnen & b2);
30       Birnen operator - (Birnen & b2);
31
32       // get- und set-Unterprogramme fuer das wert-Attribut:
33       ganz getWert() const; // Liefert den Wert dieses Birnen-Objektes
34       ganz setWert(ganz n); // Liefert den alten Wert dieses Birnen-Objektes
35                             // und setzt n als neuen Wert fest.
36    // ------------------------------------------------------------------------
37    // Ein Freund dieser Klasse (ein Ausgabeoperator fuer Birnen-Objekte):
38       friend ostream& operator<< (ostream & ausgabe, const Birnen & zahl);
39 }; // class Birnen
40 // ---------------------------------------------------------------------------
41 #endif // #ifndef Birnen_h

42 // Datei Birnen.cpp
43 /* ------------------------------------------------------------------------
44 Implementierungsdatei zur Kopfdatei Birnen.h. Enthaelt die Definitionen
45 der Methoden der Klasse Birnen und die Definition des befreundeten Ausgabe-
46 Operators <<.
47 ------------------------------------------------------------------------ */
48 #include "Birnen.h"
49 #include <iostream>
50 using namespace std;
51 // ---------------------------------------------------------------------------
52 Birnen::Birnen(ganz g) {
53    setWert(g);
54 } // Allgemeiner und Standard-Konstruktor
55 // ---------------------------------------------------------------------------
56 Birnen Birnen::operator + (Birnen & b2) {
57    if (wert == ERR || b2.wert == ERR) return Birnen(ERR);
58    return Birnen(wert + b2.wert);
59 } // operator +
60 // ---------------------------------------------------------------------------
61 Birnen Birnen::operator - (Birnen & b2) {
62    if (wert == ERR || b2.wert == ERR) return Birnen(ERR);
63    return Birnen(wert - b2.wert);
64 } // operator -
65 // ---------------------------------------------------------------------------
66 Birnen::ganz Birnen::getWert() const {
67    return wert;
68 } // getWert
69 // ---------------------------------------------------------------------------
70 Birnen::ganz Birnen::setWert(ganz w) {
71    ganz erg = wert;
```

```
72     wert = (w < MIN || MAX < w) ? ERR : w;
73     return erg;
74  } // setWert
75  // ------------------------------------------------------------------------
76  // Ein Freund der Klasse Birnen (ein Ausgabeoperator fuer Birnen-Objekte):
77  ostream & operator << (ostream & ausgabe, const Birnen & zahl) {
78     if (zahl.wert == Birnen::ERR) {     // Statt "umstaendlich" mit
79        ausgabe << "ERR-vom-Typ-Birnen"; // zahl.getWert() darf ein Freund
80     } else {                            // direkt mit
81        ausgabe << zahl.wert;            // zahl.wert auf das wert-Attribut
82     }                                   // eines Birnen-Objektes zahl
83     return ausgabe;                     // zugreifen.
84  } // operator <<
85  // ------------------------------------------------------------------------
```

```
86  // Datei BirnenTst.cpp
87  /* ------------------------------------------------------------------------
88  Kleiner Test der Klasse Birnen. Es werden mehrere Birnen-Objekte definiert,
89  addiert und subtrahiert und die Objekte und die Rechenergebnisse werden zur
90  Standardausgabe ausgegeben.
91  --------------------------------------------------------------------- */
92  #include "Birnen.h"
93  #include <iostream> // Nicht noetig (schon in Birnen.h), aber moeglich
94  using namespace std;
95  // ------------------------------------------------------------------------
96  int main() {
97     cout << "BirnenTst: Jetzt geht es los!" << endl;
98
99     // Einige Birnen-Objekte definieren und initialisieren:
100                        // (Anfangs-) Wert der Variablen
101    Birnen b1(+800);     // 800
102    Birnen b2(-500);     // -500
103    Birnen b3( 150);     // 150
104    Birnen b4(+801);     // ERR-vom-Typ-Birnen
105    Birnen b5(-501);     // ERR-vom-Typ-Birnen
106    Birnen b6;           // ERR-vom-Typ-Birnen
107    Birnen b7(b2 + b3);  // -350
108    Birnen b8(b5 + b3);  // ERR-vom-Typ-Birnen
109
110    // Die Birnen-Objekte und einige Rechenergebnisse ausgeben:
111    cout << "b1          : " << b1           << endl;
112    cout << "b2          : " << b2           << endl;
113    cout << "b3          : " << b3           << endl;
114    cout << "b4          : " << b4           << endl;
115    cout << "b5          : " << b5           << endl;
116    cout << "b6          : " << b6           << endl;
117    cout << "b7          : " << b7           << endl;
118    cout << "b8          : " << b8           << endl;
119
120    cout << "b3 + b3     : " << b3 + b3      << endl;
121    cout << "b3 + b3 + b3: " << b3 + b3 + b3 << endl;
122    cout << "b3 - b3     : " << b3 - b3      << endl;
123    cout << "b3 - b3 - b3: " << b3 - b3 - b3 << endl;
124    cout << "b1 + b3     : " << b1 + b3      << endl;
125    cout << "b4 - b3     : " << b4 - b3      << endl;
126
127    cout << "BirnenTst: Das war's erstmal!" << endl;
128 } // main
129 /* ------------------------------------------------------------------------
130 Ausgabe des Programms BirnenTst:
131
```

```
132 BirnenTst: Jetzt geht es los!
133 b1          : 800
134 b2          : -500
135 b3          : 150
136 b4          : ERR-vom-Typ-Birnen
137 b5          : ERR-vom-Typ-Birnen
138 b6          : ERR-vom-Typ-Birnen
139 b7          : -350
140 b8          : ERR-vom-Typ-Birnen
141 b3 + b3     : 300
142 b3 + b3 + b3: 450
143 b3 - b3     : 0
144 b3 - b3 - b3: -150
145 b1 + b3     : ERR-vom-Typ-Birnen
146 b4 - b3     : ERR-vom-Typ-Birnen
147 BirnenTst: Das war's erstmal!
148 ------------------------------------------------------------------------ */
```

Aufgabe: Zum Typ `Birnen` gehören alle Ganzzahlen zwischen `Birnen::MIN` und `Birnen::-MAX`. Im Prinzip kann man die Werte von `MIN` und `MAX` beliebig festlegen, muss dabei aber die Grenzen des vordefinierten Typs `ganz` (alias `int`, siehe `Birnen.h`, Zeile ***24***) berücksichtigen. Sei der Einfachheit einmal angenommen, dass zum Typ `ganz` alle Ganzzahlen zwischen `-10_001` und `+10_000` gehören. Wie groß dürfen wir `MIN` und `MAX` höchstens wählen, wenn wir `Birnen`-Werte nur ***addieren*** und ***subtrahieren*** wollen? Wie groß dürfen wir `MIN` und `MAX` höchstens wählen, wenn wir `Birnen`-Werte auch ***multiplizieren*** und ***dividieren*** wollen?

12.5 Einfaches Erben, virtuelle und nicht-virtuelle Methoden

In C++ unterscheidet man aus Effizienzgründen zwischen ***virtuellen*** und ***nicht-virtuellen*** Objektmethoden einer Klasse. In vielen Fällen spielt der Unterschied ***keine*** Rolle. In anderen Fällen genügt die einfache Daumenregel: ***Nicht-virtuelle*** Methoden sind ein bisschen ***schneller*** und ***virtuelle*** Methoden ein bisschen ***korrekter*** (d. h. ***virtuelle Methoden*** verhalten sich eher so, wie man es erwartet). In ***Java*** hat man auf die etwas verzwickte Unterscheidung zwischen ***virtuellen*** und ***nicht-virtuellen*** Methoden verzichtet und behandelt alle Methoden wie ***virtuelle*** Methoden in ***C++***.

Virtuelle Methoden sind höchstens dann wichtig, wenn man folgende Konzepte kombiniert anwendet: ***Vererbung***, ***überschriebene Methoden*** und ***Adressen von Objekten***. Hier die Einzelheiten:

Angenommen, eine Klasse `Tief` erbt von einer Klasse `Hoch` und überschreibt dann eine geerbte Objektmethode `m`.

Das bedeutet anschaulich (so kann man sich zumindest vorstellen): Der Bauplan `Tief` enthält dann ***zwei*** Methoden `m`, die ***geerbte*** und "seine eigene", und in jedes `Tief`-***Objekt*** werden ***beide*** Methoden eingebaut, "unten" die ***geerbte*** Methode `m` und "darüber" die in `Tief` vereinbarte Methode `m`. Normalerweise wird nur die "oben liegende" Methode (die aus `Tief`) benutzt. Nur in ganz bestimmten Situationen (siehe unten) wird auf die darunter liegende Methode `m` (die von `Hoch` geerbte) zugegriffen.

Sei weiter angenommen, dass wir eine Variable `a` vom Typ ***Adresse von Hoch*** vereinbaren, diesen Adresse aber auf ein Objekt der Klasse `Tief` zeigen lassen. Das ist erlaubt, weil jedes `Tief`-***Objekt*** "alle Eigenschaften eines `Hoch`-***Objekts***" (und meistens noch ein paar mehr) hat. Dann hat die Adresse `a` ***zwei Zieltypen***, den ***statischen*** Zieltyp `Hoch` und den ***dynamischen*** Zieltyp `Tief`. Der statische Zieltyp wurde bei der ***Vereinbarung*** von `a` festgelegt und kann ***nicht*** verändert werden. Mit dem ***dynamischen*** Zieltyp ist der Typ des Objekts `*a` gemeint, auf das `a` gerade zeigt. Den dynamischen Zieltyp kann man ***verändern***, indem man `a` auf ein anderes Objekt zeigen lässt (z. B. auf ein `Hoch`-***Objekt***).

Betrachten wir jetzt den Methodenaufruf `a->m(...)`.Welche Methode `m` wird dadurch aufgerufen? Die in `Hoch` vereinbarte oder die in `Tief` vereinbarte?

Falls `m` in Hoch als ***virtuelle*** Methode vereinbart wurde, bezeichnet `a->m(...)` die in `Tief` vereinbarte Methode. In diesem Fall ist der ***dynamische*** Typ von `a` (nämlich `Tief`) entscheidend. Falls `m` in `Hoch` als ***nicht-virtuelle*** Methode vereinbart wurde, bezeichnet `a->m(...)` die in `Hoch` vereinbarte Methode. In diesem Fall ist der ***statische*** Typ von `a` (nämlich `Hoch`) entscheidend.

Folgende Begriffe gehören also zusammen:

- ***Virtuelle*** Methode, ***dynamischer*** Typ der Adresse, Methode aus der ***Unterklasse***.
- ***Nicht-Virtuelle*** Methode, ***statischer*** Typ der Adresse, Methode aus der ***Oberklasse***.

Im folgenden Beispielprogramm ist die Variable `hoti1` ("Hoch/Tief 1") vom Typ ***Adresse von Hoch***, zeigt aber auf ein `Tief`-Objekt. In der Klasse `Hoch` werden zwei Objektmethoden namens `virginia` und `nicoline` vereinbart. Die Klasse `Tief` erbt von Hoch und ***überschreibt*** diese beiden Methoden. Die Methoden namens `virginia` sind ***virtuell*** und die Methoden namens `nicoline` sind ***nicht-virtuell***.

```
1   // Datei Virtuell01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert den Unterschied zwischen virtuellen und nicht-virtuellen
4   Methoden. Die Funktionen namens virginia sind virtuell, die Funktio-
5   nen namens nicoline sind nicht-virtuell.
6   Sei Hoch eine Oberklasse von Tief und hoti eine Variable des Typs Adresse
7   von Hoch, die momentan auf ein Objekt der Klasse Tief zeigt (hoti hat den
8   statischen Zieltyp Hoch, aber den dynamischen Zieltyp Tief). Greift man
9   ueber hoti auf eine virtuelle Methode zu, bekommt man die der Klasse Tief
10  (der dynamische Zieltyp von hoti ist entscheidend). Greift man ueber hoti
11  auf eine nicht-virtuelle Methode zu, bekommt man die der Klasse Hoch (der
12  statische Zieltyp von hoti ist entscheidend). Entsprechendes gilt, wenn
13  hoti eine Referenzvariable ist.
14  ------------------------------------------------------------------------ */
15  #include <iostream>
16  #include <string>
17  using namespace std;
18  // ------------------------------------------------------------------------
19  class Hoch {
20  public:
21     virtual
22     string virginia() {
23        return "virginia aus Klasse Hoch!";
24     }
25     string nicoline() {
26        return "nicoline aus Klasse Hoch!";
27     }
28  }; // class Hoch
29  // ------------------------------------------------------------------------
30  class Tief: public Hoch {
31  public:
32     virtual
33     string virginia() {
34        return "virginia aus Klasse Tief!";
35     }
36     string nicoline() {
37        return "nicoline aus Klasse Tief!";
38     }
39  }; // class Tief
40  // ------------------------------------------------------------------------
```

```
41  int main() {
42     cout << "Virtuell01: Jetzt geht es los!" << endl;
43
44     Hoch * hoho1 = new Hoch(); // Adresse von Hoch zeigt     auf Hoch
45     Hoch * hoti1 = new Tief(); // Adresse von Hoch  zeigt     auf Tief
46     Tief * titi1 = new Tief(); // Adresse von Tief  zeigt     auf Tief
47     Tief   titi2;              // Normale Variable des Typs Tief
48     Hoch & hoti2 = titi2;      // Referenz auf Hoch referiert auf Tief
49
50     cout << "hoho1->nicoline(): " << hoho1->nicoline() << endl;
51     cout << "hoho1->virginia(): " << hoho1->virginia() << endl;
52
53     cout << "titi1->nicoline(): " << titi1->nicoline() << endl;
54     cout << "titi1->virginia(): " << titi1->virginia() << endl;
55
56     cout << "hoti1->nicoline(): " << hoti1->nicoline() << endl;
57     cout << "hoti1->virginia(): " << hoti1->virginia() << endl;
58
59     cout << "hoti2. nicoline(): " << hoti2.nicoline () << endl;
60     cout << "hoti2. virginia(): " << hoti2.virginia () << endl;
61
62     cout << "titi2. nicoline(): " << titi2.nicoline () << endl;
63     cout << "titi2. virginia(): " << titi2.virginia () << endl;
64
65     cout << "Virtuell01: Das war's erstmal!" << endl;
66  } // main
67  /* ------------------------------------------------------------------------
68  Ausgabe des Programms Virtuell01:
69
70  Virtuell01: Jetzt geht es los!
71  hoho1->nicoline(): nicoline aus Klasse Hoch!
72  hoho1->virginia(): virginia aus Klasse Hoch!
73  titi1->nicoline(): nicoline aus Klasse Tief!
74  titi1->virginia(): virginia aus Klasse Tief!
75  hoti1->nicoline(): nicoline aus Klasse Hoch!
76  hoti1->virginia(): virginia aus Klasse Tief!
77  hoti2. nicoline(): nicoline aus Klasse Hoch!
78  hoti2. virginia(): virginia aus Klasse Tief!
79  titi2. nicoline(): nicoline aus Klasse Tief!
80  titi2. virginia(): virginia aus Klasse Tief!
81  Virtuell01: Das war's erstmal!
82  ------------------------------------------------------------------------ */
```

Wenn einem die Unterscheidung zwischen ***virtuellen*** und ***nicht-virtuellen*** Methoden zu kompliziert ist, sollte man erst mal versuchen, ***alle*** Methoden ***virtuell*** zu machen (wie in ***Java***). ***Destruktoren*** müssen in aller Regel ***virtuell*** deklariert sein (wenn gleichzeitig auch Vererbung und Adressen eine Rolle spielen). Können Sie erklären, warum?

12.6 Einfaches Erben, eine kleine Klassenhierarchie

```
1   // Datei Figuren01.cpp
2   /* ------------------------------------------------------------------------
3   Eine kleine Hierarchie von Klassen (Unterklassen sind eingerueckt):
4
5   Punkt
6      Quadrat
7         Rechteck
8      Kreis
9
10  Der Einfachheit halber werden alle Klassen in dieser einen Datei
11  spezifiziert und implementiert (deklariert und definiert).
12  ------------------------------------------------------------------------ */
13  #include <iostream>
14  #include <cmath>       // fuer sqrt()
15  #include <string>
16  #include <sstream>     // fuer ostringstream (mit neuen string-Objekten)
17  using namespace std;
18  // ------------------------------------------------------------------------
19  class Punkt {
20  // Jedes Objekt dieser Klasse stellt einen Punkt auf einer Ebene dar.
21  private:
22     double x, y; // Die Koordinaten des Punktes
23  public:
24     // Ein (Standard- und allgemeiner) Konstruktor:
25     Punkt(double x=0, double y=0);
26
27     // Methoden:
28             double urAbstand (); // Liefert den Abstand vom Ursprung
29             void   urSpiegeln(); // Spiegelt diesen Punkt am Ursprung
30     virtual string text      () const; // Liefert die Koordinaten
31     virtual string toString  () const; // Liefert eine Beschreibung
32  }; // class Punkt
33  // ------------------------------------------------------------------------
34  class Quadrat: public Punkt {
35  // Jedes Objekt dieser Klasse stellt ein Quadrat auf einer Ebene dar. Die
36  // Seiten eines solchen Quadrats verlaufen parallel zur x- bzw. y-Achse.
37  protected:
38     double deltaX; // Abstand der x-Seite vom Zentrum
39  public:
40     // Ein (Standard- und allgemeiner) Konstruktor:
41     Quadrat(double x=0, double y=0, double deltaX=0.5);
42
43     // Methoden (eine):
44     virtual string toString() const; // Liefert eine Beschreibung
45  }; // class Quadrat
46  // ------------------------------------------------------------------------
47  class Rechteck: public Quadrat {
48  // Jedes Objekt dieser Klasse stellt ein Rechteck auf einer Ebene dar. Die
49  // Seiten eines solchen Rechtecks verlaufen parallel zur x- bzw. y-Achse.
50  protected:
51     double deltaY; // Abstand der y-Seite vom Zentrum
52  public:
53     // Ein (Standard- und allgemeiner) Konstruktor:
54     Rechteck(double x=0, double y=0, double deltaX=0.5, double deltaY=0.5);
55
56     // Methoden (eine):
57     string virtual toString() const ; // Liefert eine Beschreibung
58  }; // class Rechteck
59  // ------------------------------------------------------------------------
```

```
60  class Kreis: public Punkt {
61  // Jedes Objekt dieser Klasse stellt einen Kreis in einer Ebene dar.
62  protected:
63     double radius; // Der Radius dieses Kreises
64  public:
65     // Ein (Standard- und allgemeiner) Konstruktor:
66     Kreis(double x=0, double y=0, double radius=1);
67
68     // Methoden (eine):
69     virtual string toString() const ; // Liefert eine Beschreibung
70  }; // class Kreis
71  // ------------------------------------------------------------------------
72  // Definitionen der Methoden der Klasse Punkt:
73
74          Punkt::Punkt(double x, double y) {this->x = x; this->y = y;}
75  double Punkt::urAbstand ()   {return sqrt(x*x + y*y);}
76  void   Punkt::urSpiegeln()   {x = -x; y = -y;}
77
78  string Punkt::text() const {
79     ostringstream ost;
80     ost << "(" << x << ", " << y << ")";
81     return ost.str();
82  } // text
83
84  string Punkt::toString() const  {
85     return "Punkt bei " + this->text();
86  } // toString
87  // ------------------------------------------------------------------------
88  // Definitionen der Methoden der Klasse Quadrat:
89
90  Quadrat::Quadrat(double x, double y, double deltaX) :
91     Punkt(x, y)              // Den "Punkt im Quadrat"    initialisieren
92  {
93     this->deltaX = deltaX;  // Das zusaetzliche Attribut initialisieren
94  } // Konstruktor Quadrat
95
96  string Quadrat::toString() const {
97     ostringstream ost;
98     ost << "Quadrat,  Zentrum bei " << this->text() <<
99        ", Seite: " << 2*deltaX;
100    return ost.str();
101 } // toString
102 // ------------------------------------------------------------------------
103 // Definitionen der Methoden der Klasse Rechteck:
104
105 Rechteck::Rechteck(double x,      double y,
106                    double deltaX, double deltaY) :
107    Quadrat(x, y, deltaX)   // Das "Quadrat im Rechteck" initialisieren
108 {
109    this->deltaY = deltaY;  // Das zusaetzliche Attribut initialisieren
110 } // Konstruktor Rechteck
111
112 string Rechteck::toString() const {
113    ostringstream ost;
114    ost << "Rechteck, Zentrum bei " << this->text() <<
115       ", x-Seite: " << 2*deltaX << ", y-Seite: " << 2*deltaY;
116    return ost.str();
117 } // toString
118 // ------------------------------------------------------------------------
119 // Definitionen der Methoden der Klasse Quadrat:
120
121 Kreis::Kreis(double x, double y, double radius) :
```

```
122    Punkt(x, y)              // Den "Punkt im Kreis"    initialisieren
123 {
124    this->radius = radius;  // Das zusaetzliche Attribut initialisieren
125 } // Konstruktor Kreis
126
127 string Kreis::toString() const {
128    ostringstream ost;
129    ost << "Kreis,    Zentrum bei " << this->text() <<
130       ", Radius: " << radius;
131    return ost.str();
132 } // toString
133 // ------------------------------------------------------------------------
134 int main() {
135    // Dient zum Testen der Klassen Punkt, Quadrat, Rechteck und Kreis.
136
137    cout << "Programm Figuren01: Jetzt geht es los!" << endl << endl;
138
139    Punkt p1(2.5, 3.4);
140    cout << "p1.text()    : " << p1.text()           << endl;
141    cout << "p1.toString(): " << p1.toString()       << endl << endl;
142
143    Quadrat q1(3.2, 4.5, 2.0);
144    cout << "q1.text()    : " << q1.text()           << endl;
145    cout << "q1.toString(): " << q1.toString()       << endl << endl;
146
147    Rechteck r1(3.1, 2.1, 4.1, 1.1 );
148    cout << "r1.text()    : " << r1.text()           << endl;
149    cout << "r1.toString(): " << r1.toString()       << endl << endl;
150
151    Kreis k1(3.3, 4.4, 2.5);
152    cout << "k1.text()    : " << k1.text()           << endl;
153    cout << "k1.toString(): " << k1.toString()       << endl << endl;
154
155    Punkt & pr = r1;
156    cout << "pr.text()    : " << pr.text()           << endl;
157    cout << "pr.toString(): " << pr.toString()       << endl << endl;
158
159    cout << "Programm Figuren01: Das war's erstmal!" << endl;
160 } // main
161 /* ------------------------------------------------------------------------
162 Ausgabe des Programms Figuren01:
163
164 Programm Figuren01: Jetzt geht es los!
165
166 p1.text()    : (2.5, 3.4)
167 p1.toString(): Punkt bei (2.5, 3.4)
168
169 q1.text()    : (3.2, 4.5)
170 q1.toString(): Quadrat,  Zentrum bei (3.2, 4.5), Seite: 4
171
172 r1.text()    : (3.1, 2.1)
173 r1.toString(): Rechteck, Zentrum bei (3.1, 2.1), x-Seite: 8.2, y-Seite: 2.2
174
175 k1.text()    : (3.3, 4.4)
176 k1.toString(): Kreis,    Zentrum bei (3.3, 4.4), Radius: 2.5
177
178 pr.text()    : (3.1, 2.1)
179 pr.toString(): Rechteck, Zentrum bei (3.1, 2.1), x-Seite: 8.2, y-Seite: 2.2
180
181 Programm Figuren01: Das war's erstmal!
182 ---------------------------------------------------------------------- */
```

12.7 Mehrfaches Erben

In ***Java*** erbt jede Klasse von genau ***einer*** anderen Klasse. Die einzige Ausnahme ist die Klasse ***Object***, die von ***null*** Klassen ("von keiner Klasse") erbt.

In ***C++*** kann jede Klasse von ***null*** Klassen, von ***einer*** Klasse oder sogar von ***mehreren*** Klassen erben. Einige Probleme lassen sich mit dem Mittel des ***mehrfachen Erbens*** besonders elegant lösen. Andererseits sind Klassenhierarchien, in denen ***mehrfaches Erben*** vorkommt, relativ schwer zu durchschauen und bieten dem Programmierer ganz besondere Möglichkeiten, schwer zu findende Fehler zu machen. In ***Java*** hat man deshalb mehrfaches Erben ***ausgeschlossen*** und als eine Art Ersatz dafür die (einfacheren und weniger fehlerträchtigen) ***Schnittstellen*** (interfaces) eingeführt.

Anmerkung: In diesem Skript wird der Begriff ***mehrfaches Erben*** (oder ***mehrfaches Beerben***, mit Dank an Christoph Knabe) anstelle der verbreiteten Bezeichnung ***mehrfache Vererbung*** verwendet. Die verbreitete Bezeichnung wird normalerweise nicht in ihrer wörtlichen Bedeutung verwendet (in allen OO-Programmiersprachen darf eine Klasse ihre Elemente ***mehrfach*** an andere Klassen ***vererben***, aber das ist mit ***mehrfacher Vererbung*** nicht gemeint), sondern in einer eher künstlichen Bedeutung ("ein Klasse K darf mehrere andere Klassen zum ***vererben*** ihrer Elemente an K auffordern"). Das englische Verb ***to inherit*** bedeutet ***erben*** und nicht ***vererben*** (vererben heisst im Englischen ***to leave***, ***to bequeath***) und somit sollte man den englischen Fachausdruck ***multiple inheritance*** auch nicht mit mehrfacher ***Ver***erbung übersetzen. Ein letztes Argument: In OO-Programmiersprachen (Smalltalk, Eiffel, C++, Java) wird immer nur das ***Erben*** konkret ***notiert***, an welche Klassen ***vererbt*** wird kann höchstens im Kommentar angegeben werden.

12.7.1 Mehrfaches Erben, ein einfaches (?) Beispiel

```
1   // Datei Mehrfach01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert einen einfachen Fall von Mehrfachbeerbung. Die Klasse Tief
4   beerbt die beiden Klassen Links und Rechts. Die Klassennamen Tief, Links
5   und Rechts sollen die Positionen der Klassen in einem Beerbungsgraphen
6   deutlich machen.
7   ------------------------------------------------------------------------ */
8   #include <iostream>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  class Links { // Eine Basisklasse von Tief ("vererbt an Tief")
12  // Jedes Objekt dieser Klasse stellt eine (rechteckige) Flaeche dar.
13  public:
14     // Ein Standard- und allgemeiner Konstruktor:
15     Links(int laenge=0, int breite=0) {
16        setLaenge(laenge);
17        setBreite(breite);
18     } // Konstruktor Links
19
20     // Methoden:
21     int  getFlaeche() const {return laenge * breite;}
22     void setLaenge(int laenge) {
23        // Statt einer negativen Laenge wird 0 genommen:
24        this->laenge = laenge > 0 ? laenge : 0;
25     } // setLaenge
26
27     void setBreite(int breite) {
28        // Statt einer negativen Breite wird 0 genommen:
29        this->breite = (breite > 0) ? breite : 0;
30     } // setBreite
31
```

```
32 protected:
33    int laenge;
34    int breite;
35 }; // class Links
36 // ------------------------------------------------------------------------
37 class Rechts { // Noch eine Basisklasse von Tief
38 // Jedes Objekt dieser Klasse stellt ein Gewicht dar.
39 public:
40    // Ein (Standard-) Konstruktor:
41    Rechts(double gewicht=0) {
42       setGewicht(gewicht);
43    } // Konstruktor Rechts
44
45    // Methoden:
46    double getGewicht() const {return gewicht;}
47
48    void setGewicht(double gewicht) {
49       // Statt eines negativen Gewichts wird 0 genommen:
50       this->gewicht = gewicht > 0 ? gewicht : 0;
51    } // setGewicht
52
53 protected:
54    double gewicht;
55 }; // class Rechts
56 // ------------------------------------------------------------------------
57 class Tief: public Links, public Rechts {
58 // Jedes Objekt dieser Klasse stellt eine Flaeche dar, die mit einem
59 // bestimmten Gewicht belastet ist.
60 public:
61    // Ein Standard- und allgemeiner Konstruktor:
62    Tief(int laenge=0, int breite=0, double gewicht=0) :
63       Links(laenge, breite), Rechts(gewicht) {
64    } // Konstruktor Tief
65
66    // Methoden:
67    double druck() const {
68       // Liefert den Druck (Gewicht pro Flaeche) dieses Objekts:
69       return gewicht / getFlaeche();
70    } // druck
71 }; // class Tief
72 // ------------------------------------------------------------------------
73 int main() {
74    // Dient zum Testen der Klasse Tief:
75    cout << "Programm Mehrfach01: Jetzt geht es los!"  << endl << endl;
76    Tief otto(12, 5, 150.0);
77    Tief emil;
78    Tief karl(10, 20, -5.0);
79
80    cout << "otto.getGewicht(): " << otto.getGewicht() << endl;
81    cout << "otto.getFlaeche(): " << otto.getFlaeche() << endl;
82    cout << "otto.druck     (): " << otto.druck     () << endl << endl;
83
84    cout << "emil.getGewicht(): " << emil.getGewicht() << endl;
85    cout << "emil.getFlaeche(): " << emil.getFlaeche() << endl;
86    cout << "emil.druck     (): " << emil.druck     () << endl << endl;
87
88    cout << "karl.getGewicht(): " << karl.getGewicht() << endl;
89    cout << "karl.getFlaeche(): " << karl.getFlaeche() << endl;
90    cout << "karl.druck     (): " << karl.druck     () << endl << endl;
91
92    cout << "Programm Mehrfach01: Das war's erstmal!"  << endl;
93 } // main
```

```
94  /* ------------------------------------------------------------------------
95  Ausgabe des Programms Mehrfach01:
96
97  Programm Mehrfach01: Jetzt geht es los!
98
99  otto.getGewicht(): 150
100 otto.getFlaeche(): 60
101 otto.druck     (): 2.5
102
103 emil.getGewicht(): 0
104 emil.getFlaeche(): 0
105 emil.druck     (): -NaN
106
107 karl.getGewicht(): 0
108 karl.getFlaeche(): 200
109 karl.druck     (): 0
110
111 Programm Mehrfach01: Das war's erstmal!
112 ------------------------------------------------------------------------ */
```

12.7.2 Mehrfaches Erben, virtuelle, überladene und überschriebene Methoden

```
1   // Datei Mehrfach02.cpp
2   // ------------------------------------------------------------------------
3   // Demonstriert mehrfache Beerbung (die Klasse Tief erbt von Links und
4   // Rechts), das Ueberladen und Ueberschreiben von Funktionsnamen (virginia,
5   // nicoline und nikolaus) und eindeutige/mehrdeutige Funktionsaufrufe.
6   // Die Funktion virginia ist virtuell, nicoline und nikolaus sind *nicht*
7   // virtuell, nicoline und nikolaus gibt es jeweils *ohne* Parameter bzw.
8   // mit einem int-Parameter.
9   // Innerhalb einer Klasse genuegt es, wenn gleichnamige Funktionen sich
10  // durch ihre Parameter unterscheiden (siehe nicoline in Klasse Tief). Bei
11  // Funktionen aus verschiedenen Klassen genuegt das nicht (siehe nikolaus
12  // aus Klasse Links und nikolaus aus Klasse Rechts).
13  // ------------------------------------------------------------------------
14  #include <iostream>
15  #include <string>
16  using namespace std;
17  // ------------------------------------------------------------------------
18  class Links { // Eine Basisklasse von Tief
19  public:
20     virtual
21     string virginia() {
22        return "virginia aus Klasse Links!";
23     }
24     string nicoline() {
25        return "nicoline aus Klasse Links!";
26     }
27     string nikolaus() {
28        return "nikolaus aus Klasse Links!";
29     }
30  }; // class Links
31  // ------------------------------------------------------------------------
32  class Rechts { // Noch eine Basisklasse von Tief
33  public:
34     virtual
35     string virginia() {
36        return "virginia aus Klasse Rechts!";
37     }
38     string nicoline() {
39        return "nicoline aus Klasse Rechts!";
40     }
```

```
41     string nikolaus(int i) {
42        return "nikolaus aus Klasse Rechts!";
43     }
44  }; // class Rechts
45  // ------------------------------------------------------------------------
46  class Tief: public Links, public Rechts {
47  public:
48     virtual
49     string virginia() {
50        return "virginia aus Klasse Tief!";
51     }
52     string nicoline() {
53        return "nicoline aus Klasse Tief!";
54     }
55     string nicoline(int i) {
56        return "nicoline mit int-Param!";
57     }
58  }; // class Tief
59  // ------------------------------------------------------------------------
60  int main() {
61     // Dient zum Testen der Klasse Tief:
62
63     // Drei Adressvariablen zeigen auf dasselbe Objekt (der Klasse Tief):
64     Tief   * at = new Tief();
65     Links  * al = at;
66     Rechts * ar = at;
67
68     cout << "Mehrfach02: Jetzt geht es los!"                      << endl;
69     cout << "virginia ist virtuell:"                              << endl;
70
71     cout << "at->        virginia(): " << at->        virginia()   << endl;
72     cout << "at->Links ::virginia(): " << at->Links ::virginia()   << endl;
73     cout << "at->Rechts::virginia(): " << at->Rechts::virginia()   << endl;
74
75     cout << "al->        virginia(): " << al->        virginia()   << endl;
76     cout << "ar->        virginia(): " << ar->        virginia()   << endl;
77
78     cout << endl << "nicoline ist nicht virtuell:"                 << endl;
79
80     cout << "at->        nicoline():  " << at->        nicoline()  << endl;
81     cout << "at->Links ::nicoline():  " << at->Links ::nicoline()  << endl;
82     cout << "at->Rechts::nicoline():  " << at->Rechts::nicoline()  << endl;
83
84     cout << "al->        nicoline():  " << al->        nicoline()  << endl;
85     cout << "ar->        nicoline():  " << ar->        nicoline()  << endl;
86
87     cout << endl << "nikolaus ist mehrdeutig:"                     << endl;
88  // cout << "at->        nikolaus():  " << at->        nikolaus()  << endl;
89     cout << "at->Links ::nikolaus():  " << at->Links ::nikolaus()  << endl;
90     cout << "at->Rechts::nikolaus(1): " << at->Rechts::nikolaus(5) << endl;
91
92     cout << endl << "nicoline ist ueberladen, aber eindeutig:"     << endl;
93     cout << "at->        nicoline():  " << at->        nicoline()  << endl;
94     cout << "at->        nicoline(5): " << at->        nicoline(5) << endl;
95
96  } // main
97  /* ------------------------------------------------------------------------
98  Fehlermeldung des Gnu-Compilers, wenn Zeile 88 kein Kommentar ist:
99
100 Mehrfach02.cpp: In function `int main(...)':
101 Mehrfach02.cpp:88: request for method `nikolaus' is ambiguous
102 ------------------------------------------------------------------------
```

```
103 Ausgabe des Programms Mehrfach02:
104
105 Mehrfach02: Jetzt geht es los!
106 virginia ist virtuell:
107 at->         virginia(): virginia aus Klasse Tief!
108 at->Links ::virginia(): virginia aus Klasse Links!
109 at->Rechts::virginia(): virginia aus Klasse Rechts!
110 al->         virginia(): virginia aus Klasse Tief!
111 ar->         virginia(): virginia aus Klasse Tief!
112
113 nicoline ist nicht virtuell:
114 at->         nicoline():  nicoline aus Klasse Tief!
115 at->Links ::nicoline():  nicoline aus Klasse Links!
116 at->Rechts::nicoline():  nicoline aus Klasse Rechts!
117 al->         nicoline():  nicoline aus Klasse Links!
118 ar->         nicoline():  nicoline aus Klasse Rechts!
119
120 nikolaus ist mehrdeutig:
121 at->Links ::nikolaus():  nikolaus aus Klasse Links!
122 at->Rechts::nikolaus(1): nikolaus aus Klasse Rechts!
123
124 nicoline ist ueberladen, aber eindeutig:
125 at->         nicoline():  nicoline aus Klasse Tief!
126 at->         nicoline(5): nicoline mit int-Param!
127 ------------------------------------------------------------------------ */
```

Rein virtuelle Methoden (pure virtual methods) entsprechen genau den ***abstrakten Methoden*** in Java. Für eine ***rein virtuelle*** Methode braucht und darf man ***keinen Rumpf*** angeben (stattdessen "***= 0***"). Eine Klasse, die mindestens ***eine*** rein virtuelle Methode enthält, wird dadurch automatisch zu einer ***abstrakten Klasse***. Von einer solchen Klasse kann man ***keine*** Objekte vereinbaren, sie kann nur als ***Oberklasse*** für weitere Klassendefinitionen dienen. Die erbenden Klassen müssen die geerbten ***rein virtuellen*** Methoden mit ***richtigen*** Methoden überschreiben, um ***konkrete*** Klassen zu sein. Hier ein kleines Beispielprogramm:

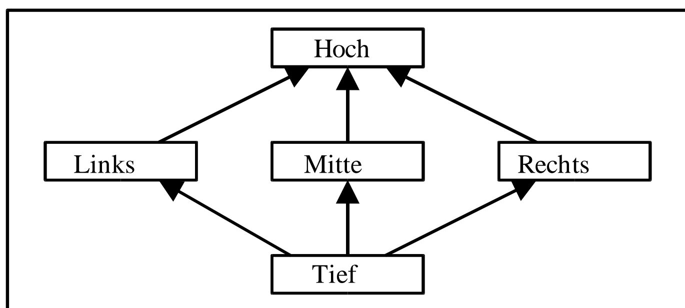
```
1   // Datei Virtuell02.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert eine Klasse Hoch die 2 rein virtuelle Methoden (pure virtual
4   methods) enthaelt. Die Klasse wird dadurch automatisch abstrakt.
5   ------------------------------------------------------------------------ */
6   #include <iostream>
7   using namespace std;
8   // ------------------------------------------------------------------------
9   // Die Klasse Hoch ist abstrakt:
10  class Hoch {
11     protected:
12        int zahl;
13     public:
14        virtual int  getZahl() const    = 0; // Eine rein virtuelle Methode
15        virtual void setZahl(int zahl) = 0; // Eine rein virtuelle Methode
16  }; // class Hoch
17  // ------------------------------------------------------------------------
18  // Die Klasse Tief erbt von Hoch, implementiert die rein virtuellen
19  // Methoden und ist somit eine konkrete Klasse:
20  class Tief: public Hoch {
21     public:
22        int  getZahl() const    {return zahl;}
23        void setZahl(int zahl) {this->zahl = zahl;}
24  }; // class Tief
25  // ------------------------------------------------------------------------
```

```
26  int main() {
27     cout << "Virtuell02: Jetzt geht es los!" << endl;
28  // Hoch h1;
29     Tief t1;
30     t1.setZahl(17);
31     cout << "t1.getZahl(): " << t1.getZahl() << endl;
32     cout << "Virtuell02: Das war's erstmal!" << endl;
33  } // main
34  /* ------------------------------------------------------------------------
35  Fehlermeldung des Gnu-Compilers, wenn Zeile 28 kein Kommentar ist:
36
37  Virtuell02.cpp:28: cannot declare variable `h1' to be of type `Hoch'
38  Virtuell02.cpp:28:   since the following virtual functions are abstract:
39  Virtuell02.cpp:15:   void Hoch::setZahl(int)
40  Virtuell02.cpp:14:   int Hoch::getZahl() const
41  ------------------------------------------------------------------------
42  Ausgabe des Programms Virtuell02:
43
44  Virtuell02: Jetzt geht es los!
45  t1.getZahl(): 17
46  Virtuell02: Das war's erstmal!
47  ---------------------------------------------------------------------- */
```

Die "konkreten" (impure?) Methoden `getZahl` und `setZahl`, mit denen die Klasse `Tief` die geerbten ("abstrakten", reinen virtuellen) Methoden überschreibt, sind automatisch virtuell, auch wenn man ihre Vereinbarungen nicht mit dem Schlüsselwort `virtual` beginnt (siehe dazu auch das Beispielprogramm `Virtuel03.cpp`, hier nicht wiedergegeben).

12.7.3 Mehrfaches Erben, die Nappo-Frage

Ein ***Nappo*** ist eine ***Raute*** (aus klebrigem Nougat, mit Schokolade überzogen). Wenn ein ***Beerbungsgraph*** Rauten enthält, sollte der Programmierer sich unbedingt die Nappo-Frage stellen (und möglichst korrekt beantworten). Hier ein Beispiel für einen Beerbungsgraphen mit einem ***mehrfachen Nappo*** darin.



Die ***Knoten*** eines Beerbungsgraphen sind (mit) ***Klassen*** (beschriftet). Ein ***Pfeil*** von A nach B bedeutet: Die Klasse A ***berbt*** die Klasse B (dargestellt wird also die ***Beerbungsrelation***, nicht die umgekehrte ***Vererbungsrelation***).

Im Beispiel bilden schon die äußeren 4 Klassen (***Hoch***, ***Links***, ***Rechts*** und ***Tief***) allein einen ***Nappo*** (eine Raute) und die Klasse ***Mitte*** macht daraus sogar einen ***mehrfachen Nappo***.

Die Nappo-Frage: ***Wie oft*** erbt die Klasse ***Tief*** (indirekt) die Elemente der Klasse ***Hoch***? Einmal? Dreimal? Zweimal?

Antwort: Das kommt darauf an. Jede der 3 Antworten (einmal, zweimal, dreimal) kann richtig sein.

Die ***Antwort*** auf die Nappo-Frage ist ***nicht*** für alle Elemente der Klasse `Hoch` gleich ***wichtig***. Sie ist ***nicht wichtig*** für alle ***Klassenelemente*** und für alle ***Methoden*** der Klasse `Hoch` (für ***Klassenmethoden*** ist die Antwort also ***doppelt unwichtig***). ***Wichtig*** ist die Antwort hauptsächlich für ***Objektattribute***. Hier eine genauere Version der Nappo-Frage: Wenn in der Klasse `Hoch` ein ***Objektattribut*** vereinbart wird, wieviele entsprechende Attribute enthält dann jedes `Tief`-***Objekt***? Eins, zwei oder drei?

```
1   // Datei Mehrfach03.cpp
2   // ------------------------------------------------------------------------
3   // Die Klasse Tief beerbt die Klasse Hoch dreimal, naemlich ueber die
4   // Klassen Links, Mitte und Rechts. Somit enthaelt jedes Tief-Objekt drei
5   // int-Variablen ("Kopien" der Variablen zahl aus der Klasse Hoch).
6   // Siehe auch die sehr aehnliche Datei Mehrfach04.cpp.
7   // ------------------------------------------------------------------------
8   #include <iostream>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  class Hoch { // Basisklasse von Links, Mitte und Rechts
12  private:
13     int zahl;
14  public:
15     Hoch(int zahl=0)            {setZahl(zahl);}
16     int  getZahl() const        {return zahl;  }
17     void setZahl(int zahl)      {this->zahl = zahl>0 ? zahl : 0;}
18  }; // class Hoch
19  // ------------------------------------------------------------------------
20  class Links: public Hoch {
21  public:
22     Links(int breite=0)         {setZahl(breite); }
23     int  getBreite() const      {return getZahl();}
24     void setBreite(int breite) {setZahl(breite); }
25  }; // class Links
26  // ------------------------------------------------------------------------
27  class Mitte: public Hoch {
28  public:
29     Mitte(int laenge=0)         {setZahl(laenge); }
30     int  getLaenge() const      {return getZahl();}
31     void setLaenge(int laenge) {setZahl(laenge); }
32  }; // class Mitte
33  // ------------------------------------------------------------------------
34  class Rechts: public Hoch {
35  public:
36     Rechts(int hoehe=0)         {setZahl(hoehe);  }
37     int  getHoehe() const       {return getZahl();}
38     void setHoehe(int hoehe)    {setZahl(hoehe);  }
39  }; // class Rechts
40  // ------------------------------------------------------------------------
41  class Tief: public Links, public Mitte, public Rechts {
42  public:
43     Tief(int breite, int laenge, int hoehe)
44        : Links(breite), Mitte(laenge), Rechts(hoehe) {}
45
46     void drucke() {
47        cout << "Ein Tief-Objekt, breite " << getBreite() <<
48                               ", laenge " << getLaenge() <<
49                                ", hoehe " << getHoehe () << endl;
50     } // drucke
51  }; // Tief
52  // ------------------------------------------------------------------------
```

```
53 int main() {
54    cout << "Mehrfach03: Jetzt geht es los!" << endl;
55    Tief objekt1(15, 40, 30);
56    objekt1.drucke();
57    objekt1.setBreite(40);
58    objekt1.setLaenge(50);
59    objekt1.setHoehe (25);
60    objekt1.drucke();
61    cout << "Mehrfach03: Das war's erstmal!" << endl;
62 } // main
63 /* ------------------------------------------------------------------------
64 Ausgabe des Programms Mehrfach03:
65
66 Mehrfach03: Jetzt geht es los!
67 Ein Tief-Objekt, breite 15, laenge 40, hoehe 30
68 Ein Tief-Objekt, breite 40, laenge 50, hoehe 25
69 Mehrfach03: Das war's erstmal!
70 ------------------------------------------------------------------------ */
```

Man beachte in der Ausgabe des Programms besonders, dass die `laenge` und die `hoehe` des `Tief`-Objekts (sowohl vor als auch nach ihrer Veränderung durch entsprechende `set`-Befehle) sich voneinander ***unterscheiden*** (`laenge 40` ist ***ungleich*** `hoehe 30`, `laenge 50` ist ***ungleich*** `hoehe 25`).

Das folgende Beispielprogramm (`Mehrfach04`) unterscheidet sich vom vorigen (`Mehrfach03`) nur dadurch, dass darin ***zweimal*** das Schlüsselwort `virtual` vorkommt (in Zeile 27 und 34). Durch dieses unscheinbare Wort wird die richtige Antwort auf die Nappo-Frage von "***dreimal***" zu "***zweimal***" verändert.

```
1  // Datei Mehrfach04.cpp
2  // ------------------------------------------------------------------------
3  // Hier beerbt die Klasse Tief die Klasse Hoch nur zweimal: einmal ueber
4  // die Klasse Links, und einmal ueber die Klassen Mitte und Rechts. Somit
5  // enthaelt jedes Tief-Objekt nur zwei int-Variablen ("Kopien" von zahl).
6  // Siehe auch die sehr aehnliche Datei Mehrfach03.cpp.
7  // ------------------------------------------------------------------------
8  #include <iostream>
9  using namespace std;
10 // ------------------------------------------------------------------------
11 class Hoch { // Basisklasse von Links, Mitte und Rechts
12 private:
13    int zahl;
14 public:
15    Hoch(int zahl=0)           {setZahl(zahl);}
16    int  getZahl() const       {return zahl;  }
17    void setZahl(int zahl)     {this->zahl = zahl>0 ? zahl : 0;}
18 }; // class Hoch
19 // ------------------------------------------------------------------------
20 class Links: public Hoch {                       // <--- nicht virtuell !!
21 public:
22    Links(int breite=0)        {setZahl(breite); }
23    int  getBreite() const     {return getZahl();}
24    void setBreite(int breite) {setZahl(breite); }
25 }; // class Links
26 // ------------------------------------------------------------------------
27 class Mitte: public virtual Hoch {               // <--- virtuell !!
28 public:
29    Mitte(int laenge=0)        {setZahl(laenge); }
30    int  getLaenge() const     {return getZahl();}
31    void setLaenge(int laenge) {setZahl(laenge); }
32 }; // class Mitte
```

```
33 // ------------------------------------------------------------------------
34 class Rechts: public virtual Hoch {            // <--- virtuell !!
35 public:
36    Rechts(int hoehe=0)         {setZahl(hoehe);  }
37    int  getHoehe() const       {return getZahl();}
38    void setHoehe(int hoehe)    {setZahl(hoehe);  }
39 }; // class Rechts
40 // ------------------------------------------------------------------------
41 class Tief: public Links, public Mitte, public Rechts {
42 public:
43    Tief(int breite, int laenge, int hoehe)
44       : Links(breite), Mitte(laenge), Rechts(hoehe) {}
45
46    void drucke() {
47       cout << "Ein Tief-Objekt, breite " << getBreite() <<
48                                ", laenge " << getLaenge() <<
49                                 ", hoehe " << getHoehe () << endl;
50    }
51 }; // class Tief
52 // ------------------------------------------------------------------------
53 int main() {
54    cout << "Mehrfach04: Jetzt geht es los!" << endl;
55    Tief objekt1(15, 40, 30);
56    objekt1.drucke();
57    objekt1.setBreite(40);
58    objekt1.setLaenge(50);
59    objekt1.setHoehe (25);
60    objekt1.drucke();
61    cout << "Mehrfach04: Das war's erstmal!" << endl;
62 } // main
63 /* ------------------------------------------------------------------------
64 Ausgabe des Programms Mehrfach04 (mit "laenge gleich hoehe"):
65
66 Mehrfach04: Jetzt geht es los!
67 Ein Tief-Objekt, breite 15, laenge 30, hoehe 30
68 Ein Tief-Objekt, breite 40, laenge 25, hoehe 25
69 Mehrfach04: Das war's erstmal!
70 ------------------------------------------------------------------------ */
```

Aufgabe: Modifizieren Sie das Programm `Mehrfach04` so, dass die richtige Antwort auf die Nappo-Frage "***einmal***" lautet.

12.8 Klassen in C++ und in Java, wichtige Unterschiede

1. In Java beerbt jede Klasse genau ***eine*** andere Klasse (Ausnahme: die Klasse `Object`) und alle Klassen bilden (mit der beerbt-Relation) einen ***Baum*** mit der Klasse `Object` an der Wurzel. In C++ kann eine Klasse beliebig viele (0, 1, 2, 3, ...) andere Klassen beerben. Deshalb bilden alle C++-Klassen keinen Baum, sondern eine Menge von ***zyklenfreien gerichteten Graphen***.

2. In C++ werden Objekte nur auf ausdrücklichen (`delete`-) Befehl des Programmierers zerstört. Vergisst der Programmierer, ein Objekt zu zerstören, bleibt es "ewig" am Leben und belegt Speicherplatz. In Java werden Objekte automatisch zerstört, wenn keine Variable mehr darauf zeigt.

3. Eine C++-Klasse kann außer beliebig vielen Konstruktoren auch einen ***Destruktor*** enthalten. Der wird immer dann aufgerufen, wenn ein Objekt der Klasse zerstört wird.

4. Das Vereinbaren der Elemente einer Klasse ist in C++ deutlich ***komplizierter*** als in Java: Bestimmte Elemente können innerhalb einer Klassendefinition nur vereinbart werden und müssen dann noch einmal (außerhalb der Klassendefinition) definiert werden, bestimmte Attribute kann man nicht einfach da initialisieren, wo man sie vereinbart.

5. In C++ wird zwischen ***virtuellen*** und ***nicht-virtuellen*** Objektmethoden unterschieden. In Java sind alle Objektmethoden virtuell (d. h. sie unterliegen der späten Bindung).

6. Den abstrakten Methoden in Java entsprechen genau die reinen virtuellen Methoden (pure virtual methods) in C++.

7. In C++ gibt es ***keine Schnittstellen*** ("total abstrakte Klassen", interfaces) wie in Java. Man kann sie aber mit Klassen, die nur reine virtuelle Methoden enthalten, nachmachen.

13 Generische Einheiten, Teil 2 (Klassenschablonen, class templates)

In C++ gibt es ***zwei Arten*** von ***Schablonen***: ***Unterprogrammschablonen*** und ***Klassenschablonen***.

Jede Instanz einer Unterprogrammschablone ist ein Unterprogramm und jede Instanz einer Klassenschablone ist (das wird jetzt hoffentlich niemanden überraschen) eine Klasse. Unterprogrammschablonen wurden im Abschnitt 10 behandelt.

13.1 Die Klassenschablone Obst

Die folgende Klassenschablone `Obst` ist eine Weiterentwicklung und verallgemeinerte Version der Klasse `Birnen` (siehe Abschnitt 12.4). Sie hat keinen ***Typparameter*** (wie die meisten Schablonen) sondern erwartet zwei Ganzzahl-Werte `MIN` und `MAX` als Schablonenparameter.

```
1   // Datei Obst.h
2   /* ------------------------------------------------------------------------
3   Obst ist eine Klassenschablone. Jede Klasse, die als Instanz dieser Scha-
4   blone erzeugt wird, ist eine Art Ganzzahltyp, zu dem eine Untergrenze MIN,
5   eine Obergrenze MAX, alle Ganzzahlen zwischen MIN und MAX und ein "undefi-
6   nierter Wert" namens ERR gehoeren.
7
8   Eine Obst-Klasse ist eine nach dieser Schablone geformte Klasse (z. B.
9   die Klasse Obst<-100, +100> oder die Klasse Obst<-500, +800> etc.).
10  Ein Obst-Objekt ist ein Objekt einer Obst-Klasse.
11
12  Diese Kopfdatei enthaelt die Definition der Klassenschablone Obst und die
13  Definitionen aller Elemente dieser Schablone.
14  ------------------------------------------------------------------------ */
15  #ifndef Obst_h // Da diese Kopfdatei Definitionen (und nicht nur Deklara-
16  #define Obst_h // tionen) enthaelt wird mehrfaches Inkludieren verhindert.
17
18  #include <iostream>
19  using namespace std;
20
21  // Alle Werte aller mit der Schablone Obst erzeugten Typen werden intern
22  // als Werte des Typs ganz realisiert:
23  typedef long int ganz;
24  // ------------------------------------------------------------------------
25  // Ein Ausgabeoperator << soll als Freund der Klasse Obst vereinbart werden.
26  // Deshalb muss vor der Definition der Klasse Obst eine entsprechende
27  // Operator-Schablone deklariert werden. Diese wiederum setzt voraus, dass
28  // vorher die Klasse Obst deklariert wird:
29
30  template<ganz MIN, ganz MAX>
31  class Obst;
32
33  template<ganz MIN, ganz MAX>
34  ostream & operator << (ostream & ausgabe, const Obst<MIN, MAX> & zahl);
35  // ========================================================================
36  // Definition einer Klassenschablone namens Obst mit 2 ganz-Parametern:
37  template<ganz MIN, ganz MAX>
38  class Obst {
39  // ---------------------------------------------
40     // Klassenelemente der Klassenschablone Obst:
41     public:
42        static ganz const ERR = MIN - 1; // Der "undefinierte Wert"
43        Obst<MIN, MAX>(ganz g = ERR);    // Standard- und allg. Konstruktor
44     // ---------------------------------------------
45     // Objektelemente der Klassenschablone Obst:
46     private:
47        ganz wert;                       // Der Wert eines Obst-Objektes
```

```
48     public:
49        // Operatoren zum Rechnen mit Obst-Objekten:
50        Obst<MIN, MAX> operator + (Obst<MIN, MAX> g2);
51        Obst<MIN, MAX> operator - (Obst<MIN, MAX> g2);
52
53        // get- und set-Methoden fuer das wert-Attribut:
54        ganz getWert() const;           // Liefert den Wert eines Obst-Objektes
55        ganz setWert(ganz w);           // Prueft und setzt den Wert
56     // ---------------------------------------------
57     // Ein Obst-Freund (ein Ausgabeoperator fuer Obst-Objekte):
58     // (die Schreibweise dieser Deklaration ist nominiert fuer den Oscar fuer
59     // die ueberraschendste Syntax in einer verbreiteten Programmierprache :-))
60        friend ostream & operator << <MIN, MAX>(
61           ostream & ausgabe, const Obst<MIN, MAX> & zahl
62        );
63     }; // class template Obst                Ende der Klassenschablone Obst
64  // ==============================================================
65  // Definitionen der Methoden der Schablone Obst und des befreundeten
66  // Ausgabeoperators << :
67
68     // Definition des Standard- und allgemeinen Konstruktors mit ganz-Param:
69     template <ganz MIN, ganz MAX>
70     Obst<MIN, MAX>::Obst(ganz g) {setWert(g);}
71
72     // Definition der Operatoren zum Rechnen mit Obst-Objekten:
73     template <ganz MIN, ganz MAX>
74     Obst<MIN, MAX> Obst<MIN, MAX>::operator + (Obst<MIN, MAX> g2) {
75        if (wert == ERR || g2.wert == ERR) return Obst<MIN, MAX>(ERR);
76        Obst<MIN, MAX> erg(wert + g2.wert);
77        return erg;
78     } // operator +
79
80     template <ganz MIN, ganz MAX>
81     Obst<MIN, MAX> Obst<MIN, MAX>::operator - (Obst<MIN, MAX> g2) {
82        if (wert == ERR || g2.wert == ERR) return Obst<MIN, MAX>(ERR);
83        Obst<MIN, MAX> erg(wert - g2.wert);
84        return erg;
85     } // operator -
86
87     // Definition der Objektmethode getWert:
88     template <ganz MIN, ganz MAX>
89     ganz Obst<MIN, MAX>::getWert() const {
90        return wert;
91     } // getWert
92
93     // Definition der Objektmethode setWert:
94     template <ganz MIN, ganz MAX>
95     ganz Obst<MIN, MAX>::setWert(ganz w) {
96        ganz erg = wert;
97        wert = (w < MIN || MAX < w) ? ERR : w;
98        return erg;
99     } // setWert
100
101    // Definition des Obst-Freundes (des Ausgabeoperators fuer Obst-Ob.):
102    template <ganz MIN, ganz MAX>
103    ostream & operator << (ostream & ausgabe, const Obst<MIN, MAX> & zahl) {
104       if (zahl.wert == Obst<MIN, MAX>::ERR) {
105          ausgabe << "ERR-vom-Typ-Obst<" << MIN << ", " << MAX << ">";
106       } else {
107          ausgabe << zahl.wert;
108       }
109       return ausgabe;
```

```
110    } // operator <<
111
112 // Jetzt sind alle Elemente der Klassenschablone Obst fertig definiert
113 // ---------------------------------------------------------------------
114
115 #endif // #ifndef Obst_h
```

Hier ein Testprogramm, in dem Objekte verschiedener `Obst`-Klassen vereinbart und bearbeitet werden:

```
1   // Datei ObstTst.cpp
2   /* ---------------------------------------------------------------------
3   Kleiner Test der Klassen-Schablone Obst. Die Namen Aepfel und Birnen werden
4   als Namen fuer verschiedene Obst-Klassen deklariert. Mehrere Aepfel- und
5   Birnen-Objekte werden definiert, addiert und subtrahiert und die Objekte
6   und die Rechenergebnisse werden zur Standardausgabe ausgegeben.
7   Es wird demonstriert, dass man Aepfel und Birnen *nicht* addieren kann.
8   ------------------------------------------------------------------- */
9   #include "Obst.h"
10  #include <iostream>
11  using namespace std;
12
13  int main() {
14     cout << "ObstTst: Jetzt geht es los!" << endl;
15
16     // Alias-Namen fuer ein paar Obst-Klassen deklarieren:
17     typedef Obst<-100, +100> Aepfel;
18     typedef Obst<-500, +800> Birnen;
19
20     // Einige Birnen-Objekte definieren und initialisieren:
21                           // (Anfangs-) Wert der Variablen
22     Birnen b1(+800);      // 800
23     Birnen b2(-500);      // -500
24     Birnen b3( 150);      // 150
25     Birnen b4(+801);      // ERR-vom-Typ-Obst<-500, 800>
26     Birnen b5(-505);      // ERR-vom-Typ-Obst<-500, 800>
27     Birnen b6;            // ERR-vom-Typ-Obst<-500, 800>
28     Birnen b7(b2 + b3);   // -350
29     Birnen b8(b5 + b3);   // ERR-vom-Typ-Obst<-500, 800>
30
31     // Die Birnen-Objekte und einige Rechenergebnisse ausgeben:
32     cout << "b1          : " << b1           << endl;
33     cout << "b2          : " << b2           << endl;
34     cout << "b3          : " << b3           << endl;
35     cout << "b4          : " << b4           << endl;
36     cout << "b5          : " << b5           << endl;
37     cout << "b6          : " << b6           << endl;
38     cout << "b7          : " << b7           << endl;
39     cout << "b8          : " << b8           << endl;
40
41     cout << "b3 + b3     : " << b3 + b3      << endl;
42     cout << "b3 + b3 + b3: " << b3 + b3 + b3 << endl;
43     cout << "b3 - b3     : " << b3 - b3      << endl;
44     cout << "b3 - b3 - b3: " << b3 - b3 - b3 << endl;
45     cout << "b1 + b3     : " << b1 + b3      << endl;
46     cout << "b4 - b3     : " << b4 - b3      << endl;
47
48     // Einige Aepfel-Objekte definieren und initialisieren:
49     Aepfel a1(60);        // 60
50     Aepfel a2(a1 + a1);   // ERR-vom-Typ-Obst<-100, 100>
51
52     // Die Aepfel-Objekte und einige Rechenergebnisse ausgeben:
```

```
53     cout << "a1           : " << a1           << endl;
54     cout << "a2           : " << a2           << endl;
55     cout << "a1 - a1      : " << a1 - a1      << endl;
56     cout << "a1 - a1 + a2: " << a1 - a1 + a2 << endl;
57
58     // "Aepfel und Birenen darf man nicht addieren" und aehnliche Typfehler:
59  // cout << (a1 + b1) << endl;
60  // b1 = a1;
61  // if (a1 == b1) cout << "Hallo!" << endl;
62
63     cout << "ObstTst: Das war's erstmal!" << endl;
64  } // main
65  /* ------------------------------------------------------------------------
66  Fehlermeldung des Borland-Compilers, wenn Zeile 59 bis 61 keine
67  Kommentare sind:
68
69  Error E2094 ObstTst.cpp 59: 'operator+' not implemented in type
70     'Obst<-100,100>' for arguments of type 'Obst<-500,800>' in function main()
71  Error E2034 ObstTst.cpp 60: Cannot convert 'Obst<-100,100>' to
72     'Obst<-500,800>' in function main()
73  Error E2094 ObstTst.cpp 61: 'operator==' not implemented in type
74     'Obst<-100,100>' for arguments of type 'Obst<-500,800>' in function main()
75  ---------------------------------------------------------------------------
76  Ausgabe des Programms ObstTst:
77
78  ObstTst: Jetzt geht es los!
79  b1           : 800
80  b2           : -500
81  b3           : 150
82  b4           : ERR-vom-Typ-Obst<-500, 800>
83  b5           : ERR-vom-Typ-Obst<-500, 800>
84  b6           : ERR-vom-Typ-Obst<-500, 800>
85  b7           : -350
86  b8           : ERR-vom-Typ-Obst<-500, 800>
87  b3 + b3      : 300
88  b3 + b3 + b3: 450
89  b3 - b3      : 0
90  b3 - b3 - b3: -150
91  b1 + b3      : ERR-vom-Typ-Obst<-500, 800>
92  b4 - b3      : ERR-vom-Typ-Obst<-500, 800>
93  a1           : 60
94  a2           : ERR-vom-Typ-Obst<-100, 100>
95  a1 - a1      : 0
96  a1 - a1 + a2: ERR-vom-Typ-Obst<-100, 100>
97  ObstTst: Das war's erstmal!
98  --------------------------------------------------------------------- */
```

13.2 Die Standard Template Library (STL)

Die C++-Standardarbibliothek besteht zum größten Teil aus der STL. Sie enthält also keine Klassen, sondern Schablonen, vor allem Klassenschablonen. Besonders wichtig sind die Behälterschablonen wie `vector`, `deque` (double ended queue), `list` etc., deren Instanzen Behälterklassen sind (deren Instanzen Behälter sind) und die Abbildungsschablonen wie `map` und `multimap`, deren Instanzen Abbildungsklassen sind (deren Instanzen Abbildungen sind).

Achtung: In Java ist ein ***Behälter*** (`container`) ein Objekt einer Unterklasse der Klasse `java.-awt.Container`. In ein solches Behälterobjekt kann man nur Grabo-Objekte wie Knöpfe, Menüs, Textfenster etc. hineintun. Ein Objekt, in das man beliebige andere Objekte hineintun kann, wird in Java als ***Sammlung*** (`collection`) und in C++ als ***Behälter*** (`container`) bezeichnet. Hier wird das Wort ***Behälter*** immer im C++-Sinn verwendet.

In C++ ist auch der Typ `string` eine Instanz einer Klassenschablone (namens `basic_string`, mit drei Typparametern namens `charT`, `traits` und `Allocator`). Diese Schablone kann man mit verschiedenen Zeichentypen (z. B. mit dem 8-Bit-Typ `char` oder mit dem 16-Bit-Typ `w_char` für den Schablonenparameter `charT`) instanziieren.

13.3 Generische Einheiten in C++ und in Java, wichtige Unterschiede

1. In C++ kann man Typen und Werte als Schablonenparameter benutzen. In Java sind nur Typen als generische Parameter erlaubt.

2. Als aktuelle Parameter für einen generischen Typparamter sind in Java nur Java-Referenztypen erlaubt, in C++ dagegen alle Typen (auch `int` und `float` etc.).

3. In C++ kann man generische Unterprogramme (Unterprogrammschablonen) unabhängig von irgendwelchen Klassen oder Klassenschablonen vereinbaren. In Java kann man Unterprogramme (auch generische) nur innerhalb von Klassen vereinbaren.

4. Wenn man in Java eine generische Klasse mit 17 verschiedenen Parametersätzen instanziiert, existiert zur Laufzeit trotzdem nur ***eine*** generische Klasse. In C++ existieren in einem entsprechenden Fall zur Laufzeit 17 Instanzen der Klassenschablone und belegen Speicherplatz.

5. Generische Einheiten in C++: Sehr mächtig, aber auch viel zu lernen. In Java: Weniger mächtig, aber auch leichter zu lernen und zu benutzen.

14 Wiederverwendbare Teile im Vergleich: Module, Klassen, Schablonen

Die Erstellung von ***wiederverwendbaren Programmteilen*** zu unterstützen war schon immer ein wichtiges Ziel bei der Entwicklung von Programmiersprachen. ***Unterprogramme***, ***Module*** (wie Modue in Modula oder wie Pakete in Ada), ***Klassen*** und ***Schablonen*** sind die wichtigsten Konstrukte, die zur Erreichung dieses Ziels erfunden wurden.

Ein ***Stapel*** (stack) ist ein Behälter, "auf den man Werte legen" kann. Zugreifen kann man jeweils nur auf den "obersten" Wert auf dem Stapel. Ein "tiefer" liegender Wert wird erst wieder zugreifbar, nachdem alle über ihm liegende Werte vom Stapel entfernt wurden (ähnlich wie bei einem Stapel von Tellern). Als ***Komponenten*** des Stapels werden die Variablen bezeichnet, die zum Speichern der Werte dienen (ganz entsprechend wie bei Reihungen, Listen und anderen Behältern auch).

In den folgenden drei Beispielprogrammen wird ein Stapel als ***Modul***, als ***Klasse*** und schliesslich als ***Schablone*** realisiert.

14.1 Der Stapel-Modul StapelM

Zur Erinnerung: Ein ***Modul*** ist Behälter für Variablen, Unterprogramme, Typen und weitere Module etc., der aus mindestens zwei Teilen besteht: Einem öffentlichen (sichtbaren, ungeschützten) und einem privaten (unsichtbaren, geschützten) Teil.

In C/C++ ist jede ***Quelldatei*** eines Programms auch ein Modul (aber viele C/C++-Programmierer sind es nicht gewohnt, ihre Quelldateien als Module zu bezeichnen). Wenn man irgendwelche Variablen, Konstanten oder Unterprogramme direkt in einer Datei `dat.cpp` definiert, dann bezeichnen wir `dat.cpp` auch als die ***Heimatdatei*** oder als den ***Heimatmodul*** dieser Größen. Das folgende Beispiel soll deutlich machen, welche Größen zum ***öffentlichen*** Teil und welche zum ***privaten*** Teil ihres Heimatmoduls gehören.

Beispiel-01: Eine Quelldatei (oder: ein Modul) namens `dat.cpp` (siehe dazu auch die Quelldateien `Static02a.cpp` und `Static02b.cpp`)

```
1   // Datei dat.cpp
2
3                  int    var01 = 1;                  // In allen Dateien   sichtbar
4   static         int    var02 = 2;                  // Nur in Heimatdatei sichtbar
5
6          const int    con01 = 1;                    // Nur in Heimatdatei sichtbar
7   static const int    con02 = 2;                    // Nur in Heimatdatei sichtbar
8   extern const int    con03 = 3;                    // In allen Dateien   sichtbar
9
10                 char * upr01() {return "upr01!";} // In allen Dateien   sichtbar
11  extern         char * upr02() {return "upr02!";} // In allen Dateien   sichtbar
12  static         char * upr03() {return "upr03!";} // Nur in Heimatdatei sichtbar
```

Im Folgenden geht es nur um Größen, die ***direkt*** in einer Datei definiert wurden (nicht um solche Größen, die z. B. in einem Unterprogramm definiert wurden, welches in einer Datei definiert wurde). Im Beispiel-01 sind alle 8 Größen direkt in der Datei (d. h. im Modul) `dat.cpp` vereinbart.

Variablen und Unterprogramme, die ***mit*** dem Schlüsselwort `static` und Konstanten, die (mit oder) ***ohne*** `static` vereinbart wurden, sind nur in ihrer Heimatdatei sichtbar. Im Beispiel gehören somit `var02`, `con01`, `con02` und `upr03` zum ***privaten*** Teil des Moduls `dat.cpp`.

Variablen und Unterprogramme, die ***ohne*** `static` und Konstanten, die ***mit*** dem Schlüsselwort `extern` vereinbart wurden, sind in allen Dateien des Programms sichtbar. Im Beispiel gehören somit `var01`, `con03`, `upr01` und `upr02` zum ***öffentlichen*** Teil des Moduls `dat.cpp`.

Anmerkung: Diese Regeln stammen aus den 1970-er Jahren. Heute würden viele Softwareingenieure es vorziehen, wenn die Regeln genau andersherum wären: Ohne spezielle Kennzeichnung sollten Variablen privat und Konstanten öffentlich sein. Um eine Variable allgemein zugänglich zu machen, oder um den Zugriff auf eine Konstante einzuschränken, sollte der Programmierer eine spezielle Kennzeichnung (mit einem geeigneten Schlüsselwort) anbringen müssen.

Anmerkung: In C++ sollte man Größen, die nur in ihrer Heimatdatei sichtbar sein sollen, nicht mit `static` kennzeichnen, sondern ihre Vereinbarungen mit einem ***namenlosen Namensraum*** umgeben (sieh dazu im folgenden Beispiel `StapelM` die Zeilen 59 bis 63).

Der Modul `StapelM` realisiert genau ***einen*** Stapel mit festgelegter Größe (***6***) und festgelegtem Komponententyp (`char`). Die Größe des Stapels wurde absichtlich so klein gewählt, um beim Testen relativ leicht einen Stapel-Überlauf provozieren zu können. Nach dem Testen könnte man die Größe dann deutlich erhöhen.

```
1  // Datei StapelM.h
2  /* ------------------------------------------------------------------------
3  Der Modul StapelM besteht aus dieser Kopfdatei StapelM.h und der Implemen-
4  tierungsdatei StapelM.cpp. Er stellt seinen Benutzern die ueblichen Unter-
5  programme push, pop und top fuer die Bearbeitung eines Stapels (von char-
6  Werten) zur Verfuegung. Der Stapel selbst ist im unsichtbaren Teil des
7  Moduls definiert und so vor direkten Zugriffen geschuetzt.
8  ------------------------------------------------------------------------ */
9  // Diese Kopfdatei enthaelt nicht nur Deklarationen, sondern auch
10 // Definitionen (der Typen Unterlauf und Ueberlauf). Deshalb muss
11 // mehrfaches Inkludieren dieser Datei verhindert werden:
12 #ifndef StapelM_h
13 #define StapelM_h
14
15 #include <string>
16 using namespace std;
17 // ------------------------------------------------------------------------
18 class Unterlauf {                    // Hat keinen definierten Konstruktor
19 public:                              // (nur den "geschenkten" Konstruktor)
20    string melde();                   // Liefert den Text einer Fehlermeldung
21 }; // class Unterlauf
22 // ------------------------------------------------------------------------
23 class Ueberlauf {                    // Hat einen definierten Konstruktor
24 public:
25    Ueberlauf(char c = '?');          // Standard- und allgemeiner Konstruktor
26    string melde();                   // Liefert den Text einer Fehlermeldung
27 private:
28    const char ueberZaehlig;          // Zeichen, welches nicht mehr auf den
29                                      // Stapel passt.
30 }; // class Ueberlauf
31 // ------------------------------------------------------------------------
32 const unsigned STAPEL_GROESSE = 6;  // Absichtlich ziemlich klein
33 // ------------------------------------------------------------------------
34 // Unterprogramme zum Bearbeiten des Stapels:
35 void push(char c) throw (Ueberlauf);
36 // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgeloest.
37 // Sonst wird der Inhalt von c auf den Stapel gelegt.
38 void pop()        throw (Unterlauf);
39 // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
40 // Sonst wird das oberste Element vom Stapel entfernt.
41 char top()        throw (Unterlauf);
42 // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
43 // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
44 // bleibt dabei unveraendert).
45 bool istLeer();
46 // Liefert true wenn der Stapel leer ist und sonst false.
47 // ------------------------------------------------------------------------
48
49 #endif  // ifndef StapelM_h
```

```
50  // Datei StapelM.cpp
51  /* ------------------------------------------------------------------------
52  Der Modul StapelM besteht aus der Kopfdatei StapelM.h und dieser Implemen-
53  tierungsdatei StapelM.cpp.
54  ------------------------------------------------------------------------ */
55  #include "StapelM.h"
56  // ------------------------------------------------------------------------
57  // Der unsichtbare (geschuetze) Teil des Moduls wird durch einen
58  // namenlosen Namensraum (anonymous namespace) realisiert:
59  namespace {
60  // Der Stapel und seine Indexvariable efi (erster freier Index):
61  char      stapel[STAPEL_GROESSE];
62  unsigned  efi = 0;                  // Anfangs ist der Stapel leer
63  } // Ende des namenlosen Namensraumes
64  // ------------------------------------------------------------------------
65  // Die folgenden Definitionen gehoeren zum sichtbaren Teil des Moduls:
66  // ------------------------------------------------------------------------
67  // Definitionen der Methoden der Klassen Unterlauf und Ueberlauf:
68
69  string Unterlauf::melde() {
70     return "Der Stapel ist leer!";
71  } // Unterlauf::melde
72  // ---------------------------------------------
73  Ueberlauf::Ueberlauf(char c) : ueberZaehlig(c) {}
74
75  string Ueberlauf::melde() {
76     const string TEXT1 = "Das Zeichen ";
77     const string TEXT2 = " passt nicht mehr auf den Stapel!";
78     return TEXT1 + ueberZaehlig + TEXT2;
79  } // Ueberlauf::melde
80  // ------------------------------------------------------------------------
81  // Definitionen der Unterprogramme zum Bearbeiten des Stapels:
82  void push(char c) throw (Ueberlauf) {
83     if (efi >= STAPEL_GROESSE) {
84        throw Ueberlauf(c);
85     }
86     stapel[efi++] = c;
87  } // push
88  // ---------------------------------------------
89  void pop() throw (Unterlauf) {
90     if (efi < 0) {
91        throw Unterlauf();
92     }
93     efi--;
94  } // pop
95  // ---------------------------------------------
96  char top() throw (Unterlauf){
97     if (efi < 1) {
98        throw Unterlauf();
99     }
100    return stapel[efi-1];
101 } // top
102 // ---------------------------------------------
103 bool istLeer() {
104    return efi <= 0;
105 } // istLeer
106 // ------------------------------------------------------------------------
```

Mit Hilfe der Datei `StapelM_Tst.cpp` (hier nicht wiedergegeben) kann man den Modul `StapelM` testen. Die ***make***-Datei `StapelM_Tst.mak` (hier nicht wiedergegeben) erleichtert das Erstellen des Testprogramms `StapelM_Tst`.

14.2 Die Stapel-Klasse StapelK

Die Klasse `StapelK` ermöglicht es dem Programmierer, beliebig viele ***Stapel*** (-Objekte) erzeugen zu lassen. Dabei kann er für jeden Stapel eine ***beliebige Größe*** festlegen. Der Komponententyp des Stapels wird dagegen durch die ***Klasse*** fest ***vorgegeben*** (und ist in diesem Beispiel gleich dem Typ `char`).

```
1   // Datei StapelK.h
2   /* ------------------------------------------------------------------------
3   Die Klasse StapelK wird in dieser Kopfdatei StapelK.h definiert. Die
4   Methoden dieser Klasse (und der gschachtelten Klassen StapelK::Ueberlauf
5   und StapelK::Unterlauf) werden in der Datei StapelK.cpp definiert.
6   Jedes Objekt dieser Klasse ist ein Stapel, auf den man char-Werte legen
7   kann. Die Groesse des Stapels kann man beim Erzeguen des Objekts angeben.
8   ------------------------------------------------------------------------ */
9   #ifndef StapelK_h
10  #define StapelK_h
11
12  #include <string>
13  using namespace std;
14  // ------------------------------------------------------------------------
15  class Stapel {
16  public:
17     // ---------------------------------------------------------------------
18     // Ein Standard- und allgemeiner Konstruktor:
19     Stapel(const unsigned groesse = 10);
20     // ---------------------------------------------------------------------
21     // Destruktor, muss sein, weil privatStapel eine Adressvariable ist:
22     ~Stapel();
23     // ---------------------------------------------------------------------
24     // Objekte der folgenden beiden Klassen werden als Ausnahmen geworfen:
25
26     class Unterlauf {            // Hat keinen definierten Konstruktor
27     public:                      // (nur den "geschenkten" Konstruktor)
28        string melde();           // Liefert den Text einer Fehlermeldung
29     }; // class Unterlauf
30     // ---------------------------------------------------------------------
31     class Ueberlauf {            // Hat einen definierten Konstruktor
32     public:
33        Ueberlauf(char c = '?'); // Standard- und allgemeiner Konstruktor
34        string melde();           // Liefert den Text einer Fehlermeldung
35     private:
36        const char ueberZaehlig; // Zeichen, welches nicht mehr auf den
37                                  // Stapel passt.
38     }; // class Ueberlauf
39     // ---------------------------------------------------------------------
40     // Die Groesse des Stapel-Objekts:
41     const unsigned GROESSE;      // Der Wert wird vom Konstruktor festgelegt
42     // ---------------------------------------------------------------------
43     // Unterprogramme zum Bearbeiten des Stapels privatStapel:
44     void push(char c) throw (Ueberlauf);
45     // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgeloest.
46     // Sonst wird der Inhalt von c auf den Stapel gelegt.
47     void pop()         throw (Stapel::Unterlauf);
48     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
49     // Sonst wird das oberste Element vom Stapel entfernt.
50     char top()         throw (Stapel::Unterlauf);
51     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
52     // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
53     // bleibt dabei unveraendert).
54     bool istLeer();
```

```
55    // Liefert true, wenn der Stapel leer ist und sonst false.
56    // ---------------------------------------------------------------------
57 private:
58    char *   privatStapel; // Zeigt auf Reihung mit GROESSE Komponenten
59    unsigned efi;          // erster freier Index von privatStapel
60 // ------------------------------------------------------------------------
61 }; // class Stapel
62
63 #endif  // ifndef StapelK_h
```

```
1  // Datei StapelK.cpp
2  /* ------------------------------------------------------------------------
3  Die Klasse StapelK wird in der Kopfdatei StapelK.h definiert. Die
4  Methoden dieser Klasse (und der gschachtelten Klassen StapelK::Ueberlauf
5  und StapelK::Unterlauf) werden in dieser Datei StapelK.cpp definiert.
6  ---------------------------------------------------------------------- */
7  #include "StapelK.h"
8  #include <string>
9  using namespace std;
10 // ------------------------------------------------------------------------
11 // Definitionen von Konstruktor und Destruktor der Klasse Stapel:
12 Stapel::Stapel(const unsigned G)
13    : GROESSE(G), efi(0), privatStapel(new char[GROESSE]) {
14 }
15
16 Stapel::~Stapel() {delete [] privatStapel;}
17 // ------------------------------------------------------------------------
18 // Definitionen fuer die geschachtelten Klassen Unterlauf und Ueberlauf:
19
20 Stapel::Ueberlauf::Ueberlauf(char c) : ueberZaehlig(c) {}
21 // ---------------------------------------------
22 string Stapel::Unterlauf::melde() {
23    return "Der Stapel ist leer!";
24 } // melde
25
26 string Stapel::Ueberlauf::melde() {
27    const string TEXT1 = "Das Zeichen ";
28    const string TEXT2 = " passt nicht mehr auf den Stapel!";
29    return TEXT1 + ueberZaehlig + TEXT2;
30 } // melde
31 // ------------------------------------------------------------------------
32 // Definitionen der Methoden zum Bearbeiten des Stapels:
33 void Stapel::push(char c) throw (Ueberlauf) {
34
35    if (efi >= GROESSE) {
36       throw Ueberlauf(c);
37    }
38    privatStapel[efi++] = c;
39 } // push
40 // ---------------------------------------------
41 void Stapel::pop()      throw (Unterlauf) {
42    if (efi < 0) {
43       throw Unterlauf();
44    }
45    efi--;
46 } // pop
47 // ---------------------------------------------
48 char Stapel::top() throw (Unterlauf){
49    if (efi < 1) {
50       throw Unterlauf();
51    }
52    return privatStapel[efi-1];
```

```
53  } // top
54  // ---------------------------------------------
55  bool Stapel::istLeer() {
56    return efi <= 0;
57  } // istLeer
58  // ------------------------------------------------------------------------
59  // Damit sind alle Elemente der Klasse Stapel definiert
```

Mit Hilfe der Datei `StapelK_Tst.cpp` (hier nicht wiedergegeben) kann man die Klasse `StapelK` testen. Die *make*-Datei `StapelK_Tst.mak` (hier nicht wiedergegeben) erleichtert das Erstellen des Testprogramms `StapelK_Tst`.

14.3 Die Stapel-Schablone StapelS

Eine ***Klassen-Schablone*** kann man (ganz ähnlich wie eine Unterprogrammschablone) ***instanziieren*** und ihr dabei ***Schablonenparameter*** übergeben (z. B. ***Typen*** oder ***Werte***). Jede Instanz einer Klassen-Schablone ist eine ***Klasse***.

Die Klassen-Schablone `StapelS` ermöglicht es dem Programmierer, beliebig viele ***Stapel*** (-Objekte) erzeugen zu lassen. Dabei kann er für jeden Stapel nicht nur ***eine beliebige Größe*** sondern auch einen ***beliebigen Komponententyp*** festlegen.

```
1   // Datei StapelS.h
2   /* ------------------------------------------------------------------------
3   Die Klassen-Schablone StapelS wird in dieser Kopfdatei StapelS.h
4   definiert. Die Methoden der Schablone (und die Methoden der geschachtelten
5   Klassen StapelS::Unterlauf und StapelS::Ueberlauf) werden in der Datei
6   StapelS.cpp definiert.
7   Jedes Objekt einer Instanz dieser Schablone ist ein Stapel. Den Typ der
8   Komponenten  dieses Stapels kann man beim Instanziieren der Schablone
9   und die Groesse des Stapels beim Erzeugen des Objekts festlegen, z. B. so:
10  StapelS<char>   meinStapel1(6); // Ein Stapel fuer 6   char-Komponenten.
11  StapelS<string> meinStapel2(8); // Ein Stapel fuer 8 string-Komponenten.
12  --------------------------------------------------------------------- */
13  #ifndef StapelS_h
14  #define StapelS_h
15
16  #include <string>
17  using namespace std;
18  // ------------------------------------------------------------------------
19  template <typename KompTyp=int>  // Schablone mit einem Typ-Parameter
20  class StapelS {                  // (vorbesetzt mit dem Typ int)
21  public:
22     // ---------------------------------------------------------------------
23     // Ein Standard- und allgemeiner Konstruktor:
24     StapelS(const unsigned groesse = 10);
25     // ---------------------------------------------------------------------
26     // Destruktor, muss sein, weil privatStapel eine Adressvariable ist:
27     ~StapelS();
28     // ---------------------------------------------------------------------
29     // Objekte der folgenden beiden Klassen werden als Ausnahmen geworfen:
30
31     class Unterlauf {            // Hat keinen definierten Konstruktor
32     public:                      // (nur den "geschenkten" Konstruktor)
33        string melde();           // Liefert den Text einer Fehlermeldung
34     }; // class Unterlauf
35     // ---------------------------------------------------------------------
36     class Ueberlauf {            // Hat einen definierten Konstruktor
37     public:
38        Ueberlauf(KompTyp e);     // Allgemeiner Konstruktor
39        string melde();           // Liefert den Text einer Fehlermeldung
```

```
40     private:
41        const
42        KompTyp ueberZaehlig;    // Element, welches nicht mehr auf den
43                                 // Stapel passt.
44     }; // class Ueberlauf
45     // ---------------------------------------------------------------------
46     // Die Groesse des Stapel-Objekts:
47     const unsigned GROESSE;      // Wird vom Konstruktor initialisiert
48     // ---------------------------------------------------------------------
49     // Methoden zum Bearbeiten des Stapels privatStapel:
50     void push(KompTyp c) throw (Ueberlauf);
51     // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgeloest.
52     // Sonst wird der Inhalt von c auf den Stapel gelegt.
53     void pop()        throw (StapelS::Unterlauf);
54     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
55     // Sonst wird das oberste Element vom Stapel entfernt.
56     KompTyp top()        throw (StapelS::Unterlauf);
57     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloest.
58     // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
59     // bleibt dabei unveraendert).
60     bool istLeer();
61     // Liefert true, falls der Stapel leer ist und sonst false.
62     // ---------------------------------------------------------------------
63  private:
64     KompTyp * privatStapel; // Zeigt auf Reihung mit GROESSE Komponenten
65     unsigned  efi;          // erster freier Index von privatStapel
66  // ------------------------------------------------------------------------
67  }; // class template StapelS
68  /* ------------------------------------------------------------------------
69  Diese Datei StapelS.h enthaelt die Definition der Schablone StapelS. Die
70  Datei StapelS.cpp enthaelt die Definitionen aller Methoden und Konstruk-
71  toren dieser Schablone. Unter dem Inklusions-Modell (ohne Schluesselwort
72  export) muessen in jeder Datei, in der die Schablone instanziiert wird,
73  die Definition der Schablone und alle zugehoerigen Definitionen enthalten
74  sein. Deshalb wird im folgenden die Definitionsdatei StapelS.cpp
75  inkludiert. Eine Anwendungsdatei muss dann nur noch diese Kopfdatei
76  StapelS.h inkludieren (und nicht etwa beide, StapelS.h und StapelS.cpp).
77  --------------------------------------------------------------------- */
78  #include "StapelS.cpp" // Um in Anwendungsdateien include-Befehle zu sparen
79
80  #endif  // ifndef StapelS_h
```

```
1   // Datei StapelS.cpp
2   /* ------------------------------------------------------------------------
3   Die Klassen-Schablone StapelS wird in der Kopfdatei StapelS.h
4   definiert. Die Methoden der Schablone (und die Methoden der geschachtelten
5   Klassen StapelS::Unterlauf und StapelS::Ueberlauf) werden in dieser Datei
6   StapelS.cpp definiert.
7   --------------------------------------------------------------------- */
8   #include <string>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  // Definitionen von Konstruktor und Destruktor der Klassen-Schablone StapelS:
12  template <typename KompTyp>
13  StapelS<KompTyp>::StapelS(const unsigned G)
14     : GROESSE(G), efi(0), privatStapel(new KompTyp[GROESSE]) {
15  }
16
17  template <class KompTyp>
18  StapelS<KompTyp>::~StapelS() {delete [] privatStapel;}
```

```
19  // ------------------------------------------------------------------------
20  // Definitionen fuer die geschachtelten Klassen Unterlauf und Ueberlauf:
21
22  template <class KompTyp>
23  StapelS<KompTyp>::Ueberlauf::Ueberlauf(KompTyp e) : ueberZaehlig(e) {}
24  // --------------------------------------------
25  template <class KompTyp>
26  string StapelS<KompTyp>::Unterlauf::melde() {
27     return "Der Stapel ist leer!";
28  } // melde
29
30  template <class KompTyp>
31  string StapelS<KompTyp>::Ueberlauf::melde() {
32     // Liefert eine nicht sehr aussagekraeftige Fehlermerldung, weil das
33     // Element ueberZaehlig (vom Typ KompTyp) hier nicht in einen String
34     // umgewandelt werden kann:
35     return "Der Stapel ist voll!";
36  } // melde
37  // ------------------------------------------------------------------------
38  // Definitionen der Methoden zum Bearbeiten des Stapels:
39  template <class KompTyp>
40  void StapelS<KompTyp>::push(KompTyp e) throw (Ueberlauf) {
41
42     if (efi >= GROESSE) {
43        throw Ueberlauf(e);
44     }
45     privatStapel[efi++] = e;
46  } // push
47  // --------------------------------------------
48  template <class KompTyp>
49  void StapelS<KompTyp>::pop() throw (Unterlauf) {
50     if (efi < 0) {
51        throw Unterlauf();
52     }
53     efi--;
54  } // pop
55  // --------------------------------------------
56  template <class KompTyp>
57  KompTyp StapelS<KompTyp>::top() throw (Unterlauf){
58     if (efi <= 0) {
59        throw Unterlauf();
60     }
61     return privatStapel[efi-1];
62  } // top
63  // --------------------------------------------
64  template <class KompTyp>
65  bool StapelS<KompTyp>::istLeer() {
66     return efi <= 0;
67  } // istLeer
68  // ------------------------------------------------------------------------
69  // Damit sind alle Elemente der Klassenschablone Stapel definiert
```

Mit Hilfe der Datei `StapelS_Tst.cpp` (hier nicht wiedergegeben) kann man die Klassen-Schablone `StapelS` testen. Die ***make***-Datei `StapelS_Tst.mak` (hier nicht wiedergegeben) erleichtert das Erstellen des Testprogramms `StapelS_Tst`.

15 Namensräume (namespaces)

Namensräume in C++ haben Ähnlichkeit mit ***Paketen*** in Java. Sie dienen dazu, ***Namenskonflikte*** zu vermeiden.

15.1 Eigene Namensräume vereinbaren

```
1   // Datei NamensRaeume01.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert Namensraeume mit Namen: Vereinbarungen einem Namensraum
4   zuordnen, auf Groessen eines Namensraumes zugreifen, using-Direktiven und
5   using-Deklarationen.
6   ------------------------------------------------------------------------ */
7   #include <string>
8   #include <iostream>
9   using namespace std;
10  // ------------------------------------------------------------------------
11  // Die folgende Variablen-Vereinbarung gehoert zum Namensraum Paket1:
12  namespace Paket1 {
13     string text = "Hallo aus dem Namensraum Paket 1 (nicht 2)!";
14  } // Paket1 (vorlaeufiges Ende)
15  // ------------------------------------------------------------------------
16  // Die folgenden beiden Vereinbarungen gehoeren zum Namensraum Paket2:
17  namespace Paket2 {
18     string text = "Hallo aus dem Namensraum Paket 2 (nicht 1)!";
19     void drucke() {
20        cout << text << endl;
21     }
22  } // Paket2
23  // ------------------------------------------------------------------------
24  // Die folgende Prozedurvereinbarung gehoert ebenfalls zu Paket1::
25  namespace Paket1 {
26     void drucke() {
27        cout << text << endl;
28     }
29  } // Paket1 (noch ein Ende)
30  // ------------------------------------------------------------------------
31  // Vereinbarungen im globalen Namensraum:
32  string text = "Hallo aus dem globalen Namensraum!";
33  //void drucke() {cout << text << endl;}
34
35  int main() {
36     using namespace Paket2; // eine using-Direktive
37     using Paket1::text;     // eine using-Deklaration
38
39     Paket1::drucke();       // drucke aus Paket1
40             drucke();       // drucke aus Paket2
41     cout << text << " bzw.aus main!" << endl; // text aus Paket?
42  } // main
43  /* ------------------------------------------------------------------------
44  Fehlermeldung des Gnu-Compilers, wenn Zeile 33 kein Kommentar ist:
45
46  NamensRaeume01.cpp:40: call of overloaded `drucke ()' is ambiguous
47  NamensRaeume01.cpp:33: candidates are: void drucke()
48  NamensRaeume01.cpp:19:                 void Paket2::drucke()
49  ------------------------------------------------------------------------
50  Ausgabe des Programms NamensRaeume01:
51
52  Hallo aus dem Namensraum Paket 1 (nicht 2)!
53  Hallo aus dem Namensraum Paket 2 (nicht 1)!
54  Hallo aus dem Namensraum Paket 1 (nicht 2)! bzw.aus main!
55  ------------------------------------------------------------------------ */
```

15.2 Eigene Namensräume, weitere using-Direktiven und -Deklarationen

```
1   // Datei NamensRaeume03.cpp
2   /* ------------------------------------------------------------------------
3   Demonstriert einen Namensraum, der using-Direktiven und using-Deklarationen
4   enthaelt. Damit kann man Vereinbarungen aus verschiedenen Namensraeumen in
5   einem Namensraum ("pauschal" bzw. "selektiv") sichtbar machen.
6   ------------------------------------------------------------------------ */
7   #include <iostream>
8   using namespace std;
9   // ------------------------------------------------------------------------
10  namespace Paket1 {
11     int var11 = 11;
12     int var12 = 12;           // Im Paket3 gibt es auch eine var12!
13     int get11() {return var11;}
14     int get12() {return var12;}
15  } // Paket1
16  // ------------------------------------------------------------------------
17  namespace Paket2 {
18     int var21 = 21;
19     int var22 = 22;
20     int get21() {return var21;}
21     int get22() {return var22;}
22  } // Paket2
23  // ------------------------------------------------------------------------
24  namespace Paket3 {
25     int var31 = 31;
26     int var12 = 32;           // Im Paket1 gibt es auch eine var12!
27  } // Paket3
28  // ------------------------------------------------------------------------
29  namespace Paket4 {
30     // Hier in Paket4 pauschal alles aus Paket1 und Paket3 sichtbar machen.
31     // Trotzdem werden Paket1::var12 und Paket3:var12 hier *nicht* sichtbar:
32     using namespace Paket1;
33     using namespace Paket3;
34     // Hier in Paket4 selektiv var21 und get22 direkt sichtbar machen (var22
35     // und get21 werden nicht direkt sichtbar):
36     using Paket2::var21;
37     using Paket2::get22;
38  } // Paket4
39  // ------------------------------------------------------------------------
40  int main() {
41     cout << "Paket4::var11  : " << Paket4::var11   << endl;
42     cout << "Paket4::get12(): " << Paket4::get12() << endl;
43     cout << "Paket4::var21  : " << Paket4::var21   << endl;
44     cout << "Paket4::get22(): " << Paket4::get22() << endl;
45  // cout << "Paket4::var22  : " << Paket4::var22   << endl;
46  // cout << "Paket4::get21(): " << Paket4::get21() << endl;
47     cout << "Paket4::var31  : " << Paket4::var31   << endl;
48  // cout << "Paket4::var12  : " << Paket4::var12   << endl;
49     cout << "Paket3::var12  : " << Paket3::var12   << endl;
50  } // main
51  /* ------------------------------------------------------------------------
52  Fehlermeldung des Gnu-Compilers, wenn die entsprechenden Zeilen keine
53  Kommentare sind
54
55  NamensRaeume03.cpp:45: `var22' undeclared in namespace `Paket4'
56  NamensRaeume03.cpp:46: `get21' undeclared in namespace `Paket4'
57  NamensRaeume03.cpp:48: use of `var12' is ambiguous
58  NamensRaeume03.cpp:12:   first declared as `int Paket1::var12' here
59  NamensRaeume03.cpp:26:   also declared as `int Paket3::var12' here
60  ------------------------------------------------------------------------
```

```
61 Ausgabe des Programms Namensraeume03:
62
63 Paket4::var11  : 11
64 Paket4::get12(): 12
65 Paket4::var21  : 21
66 Paket4::get22(): 22
67 Paket4::var31  : 31
68 Paket3::var12  : 32
69 ------------------------------------------------------------------------ */
```

15.3 Deklarationen in einem Namensraum, Definitionen außerhalb

Um eine Größe einem bestimmten ***Namensraum*** zuzuordnen, genügt es, die Größe im Namensraum zu ***deklarieren***. Die ***Definition*** kann ***außerhalb*** des Namensraumes erfolgen, z. B. in einer separaten Implementierungsdatei. Dies verhält sich ähnlich wie bei ***Klassen***: bestimmte ***Elemente*** einer Klasse kann man ***innerhalb*** der Klassen-Definition ***deklarieren*** und dann ***außerhalb definieren***.

```
1   // Datei NamensRaeume04.cpp
2   /* ------------------------------------------------------------------------
3   Deklarationen in Namensraeumen, die dazugehoerigen Definitionen koennen
4   "einzeln nachgeliefert" werden und duerfen auch in separaten Dateien stehen.
5   ------------------------------------------------------------------------ */
6   #include <iostream>
7   using namespace std;
8   // ------------------------------------------------------------------------
9   namespace Paket1 {             // Deklaration
10     extern int         var11;   // einer Variablen  (extern muss sein!)
11     extern int const const12;   // einer Konstanten (extern muss sein!)
12     extern int         get13(); // eines Unterprog. (extern kann sein!)
13  } // Paket1
14  // ------------------------------------------------------------------------
15  // Die folgenden *Definitionen* koennten auch in einer separaten Datei
16  // (oder in verschiedenen Dateien) stehen:
17  int       Paket1::  var11 = 11;
18  int const Paket1::const12 = 12;
19  int       Paket1::  get13() {var11 = 13; return var11;};
20  // ------------------------------------------------------------------------
21  int main() {
22     using namespace Paket1;
23
24     cout << "  var11  : " <<   var11   << endl;
25     cout << "const12  : " << const12   << endl;
26     cout << "  get13(): " <<   get13() << endl;
27  } // main
28  /* ------------------------------------------------------------------------
29  Ausgabe des Programms NamensRaeume04:
30
31    var11  : 11
32  const12  : 12
33    get13(): 13
34  ------------------------------------------------------------------------ */
```

15.4 Geschachtelte Namensräume

Ähnlich wie ***Pakete*** in ***Java*** (oder ***Verzeichnisse*** in einem Unix- oder Windows-***Dateisystem***) können ***Namensräume*** auch ***geschachtelt*** werden. Mit Hilfe des ***Gültigkeitsbereichsoperators*** `::` kann man auch auf geschachtelte Namensräume und ihre Größen zugreifen.

```
1   // Datei NamensRaeume05.cpp
2   /* ------------------------------------------------------------------------
3   Geschachtelte Namensraeume und ihre Sichtbarkeitsregeln.
```

```
4   ------------------------------------------------------------------------ */
5   #include <string>
6   #include <iostream>
7   using namespace std;
8   // ----------------------------------------------------------------------------
9                                         // "Voller Name"
10  string str = "String G";              // ::str
11  string get () {return str;}           // ::get ()
12  string getP();                        // ::getP()
13  string getQ();                        // ::getQ()
14
15  namespace P {                         // ::P
16     string str = "String P";           // ::P::str
17     string get () {return    str;}     // ::P::get ()
18     string getG() {return  ::str;}     // ::P::getG()
19     string getQ();                     // ::P::getQ()
20
21     namespace Q {                      // ::P::Q
22        string str = "String Q";        // ::P::Q::str
23        string get () {return      str;} // ::P::Q::get ()
24        string getG() {return ::   str;} // ::P::Q::getG()
25        string getP() {return ::P::str;} // ::P::Q::getP()
26     } // P
27  } // Q
28  // ----------------------------------------------------------------------------
29  // Definitionen fuer einige oben deklarierte Unterprogramme:
30  string    getP() {return P::   str;}
31  string    getQ() {return P::Q::str;}
32
33  string P::getQ() {return    Q::str;}
34  // ----------------------------------------------------------------------------
35  int main() {
36
37     cout << "::        get () --> " << ::get ()       << endl;
38     cout << "::        getP() --> " << ::getP()       << endl;
39     cout << "::        getQ() --> " << ::getQ()       << endl << endl;
40
41     cout << ":: P::    get () --> " << ::P::get ()    << endl;
42     cout << ":: P::    getG() --> " << ::P::getG()    << endl;
43     cout << ":: P::    getQ() --> " << ::P::getQ()    << endl << endl;
44
45     cout << ":: P:: Q:: get () --> " << ::P::Q::get () << endl;
46     cout << ":: P:: Q:: getG() --> " << ::P::Q::getG() << endl;
47     cout << ":: P:: Q:: getP() --> " << ::P::Q::getP() << endl;
48
49  } // main
50  /* ----------------------------------------------------------------------------
51  Ausgabe des Programm NamensRaeume05:
52
53  ::         get () --> String G
54  ::         getP() --> String P
55  ::         getQ() --> String Q
56
57  :: P::     get () --> String P
58  :: P::     getG() --> String G
59  :: P::     getQ() --> String Q
60
61  :: P:: Q:: get () --> String Q
62  :: P:: Q:: getG() --> String G
63  :: P:: Q:: getP() --> String P
64  ------------------------------------------------------------------------ */
```

16 Eine Schnittstelle zu einer Konsole unter Windows

In C++ sind nur relativ ***schlichte*** Ein-/Ausgabebefehle ***standardisiert***. Will man ***anspruchsvollere*** Ein-/Ausgabebefehle verwenden, um z. B. eine Grabo (***graphische Benutzeroberfläche***, engl. GUI) zu realisieren) muss man ***nicht-standardisierte*** Befehle verwenden und wird damit automatisch (mehr oder weniger) plattform***abhängig***.

Schon zur Lösung der folgenden, relativ einfachen Aufgaben, reichen die standardisierten C++Befehle ***nicht*** aus: Den ***Bildschirmzeiger*** (cursor) auf dem Bildschirm zu einer bestimmten Position ***bewegen***, von der Tastatur ein ***Zeichen*** einlesen, ***ohne*** dass der Benutzer dazu auf die ***Return***-Taste drücken muss. Ein ***Zeichen ohne Echo*** einlesen. ***Farbigen*** Text (mit Vordergrund- und Hintergrund-Farbe) zum Bildschirm ausgeben.

Mit dem folgenden Modul ***Konsole*** kann man diese Aufgaben ***lösen***, allerdings ***nur*** unter einem ***Windows***-Betriebssystem (Windows 95/98/NT/ME/2000/XP), nicht unter Linux, Solaris, OS/2 oder einem anderen Betriebssystem.

Mit einer ***Konsole*** ist häufig eine ***Tastatur*** zusammen mit einem zeichenorientierten ***Bildschirm*** (meist ***25 Zeilen*** zu je ***80 Zeichen***) gemeint.

```
1   // Datei Konsole.h
2   /* ------------------------------------------------------------------------
3   Der Modul Konsole besteht aus den Dateien Konsole.h und Konsole.cpp.
4   Mit diesem Modul (und dem Gnu-Cygnus-C++-Compiler) kann man unter
5   Windows95/98/NT
6   - das Standardausgabefenster loeschen (ClearScreen)
7   - fuer das Standardausgabefenster Farben festlegen (SetColors)
8   - einzelne Zeichen und ganze Zeichenketten zu bestimmten Stellen des
9     Standardausgabefensters ausgeben (Put)
10  - im Standardausgabefenster den Cursor positionieren (GotoXY)
11  - von der Standardeingabe einzelne Zeichen "sofort" und ohne Echo lesen,
12    (d. h. der Benutzer muss nicht auf die Return-Taste druecken und das
13    eingegebene Zeichen erscheint nicht auf dem Bildschirm (GetImmediate
14    und GetImmediateAny).
15  - die Groesse des Standardausgabefensters veraendern
16    (FenstergroesseMaximal und Fenstergroesse_80_25)
17  - zu einer Farbe zyklisch die nachfolgende Farbe bzw. die vorhergehende
18    Farbe berechnen lassen (ColorSucc und ColorPred).
19  - sich den Namen einer Farbe (als C-String) liefern lassen (ColorName).
20
21  Die bool-Funktionen liefern true, wenn "alles geklappt hat". Sie liefern
22  false, wenn ein Fehler auftrat (z. B. weil die Parameter falsch waren).
23  ------------------------------------------------------------------------ */
24  #ifndef Konsole_h
25  #define Konsole_h
26  //------------------------------------------------------------------------
27  // Groesse des Standardausgabefensters:
28  short AnzZeilen();  // Liefert die Anzahl der Zeilen
29  short AnzSpalten(); // Liefert die Anzahl der Spalten
30  //------------------------------------------------------------------------
31  // Eine Farbe ist eine 4-Bit-Binaerzahl, mit je einem Bit fuer RED, GREEN
32  // BLUE und BRIGHT. BRIGHT plus RED ist BRIGHT_RED, BLUE plus GREEN ist
33  // CYAN, BRIGHT plus BLUE plus GREEN ist BRIGHT_CYAN etc. Die Farbe
34  // bright black wird hier als GRAY bezeichnet:
35  enum Color {
36     BLACK     = 0, BLUE           = 1, GREEN         = 2, CYAN         = 3,
37     RED       = 4, MAGENTA        = 5, YELLOW        = 6, WHITE        = 7,
38     GRAY      = 8, BRIGHT_BLUE    = 9, BRIGHT_GREEN =10, BRIGHT_CYAN =11,
39     BRIGHT_RED=12, BRIGHT_MAGENTA=13, BRIGHT_YELLOW=14, BRIGHT_WHITE=15
40  };
```

```
41  //------------------------------------------------------------------------
42  bool ClearScreen (const Color  vordergrund = BRIGHT_WHITE,
43                    const Color  hintergrund = BLACK);
44  // Loescht den Bildschirm und schickt den Bildschirmzeiger (cursor) nach
45  // Hause (Spalte 0, Zeile 0). Wahlweise kann eine Farbe fuer den Vorder-
46  // grund und eine Farbe fuer den Hintergrund angegeben werden. Diese
47  // Farben gelten dann fuer alle Zellen des Standardausgabefensters und
48  // alle zukuenftigen Ausgaben zum Standardausgabefenster (bis zum naechsten
49  // Aufruf dieser Funktion ClearScreen oder einer SetColors-Funktion).
50  //------------------------------------------------------------------------
51  bool SetColors      (const Color  vordergrund = BRIGHT_WHITE,
52                       const Color  hintergrund = BLACK);
53  // Legt die Farben fuer *zukuenftige* Ausgaben zum Standardausgabefenster
54  // fest. Die Farben gelten nur fuer Ausgaben wie z. B. cout << "Hallo!";
55  // oder cout.put('X'); (Kopfdatei <iostream>), nicht aber fuer Ausgaben
56  // mit den Put-Funktionen, die in diesem Modul vereinbart werden (siehe
57  // unten). Die Farben bleiben wirksam bis zum naechsten Aufruf einer
58  // SetColors-Funktion oder der ClearScreen-Funktion.
59  //------------------------------------------------------------------------
60  bool SetColors      (const short  spalte,
61                       const short  zeile,
62                       const Color  vordergrund = BRIGHT_WHITE,
63                       const Color  hintergrund = BLACK);
64  // Legt die Hintergrundfarbe und die Vordergrundfarbe fuer *eine* Zeichen-
65  // Zelle des Standardausgabefensters fest (das Standardausgabefenster
66  // besteht aus AnzZeilen * AnzSpalten vielen Zeichen-Zellen).
67  //------------------------------------------------------------------------
68  bool SetColors      (const short  von_spalte,
69                       const short  bis_spalte,
70                       const short  von_zeile,
71                       const short  bis_zeile,
72                       const Color  vordergrund = BRIGHT_WHITE,
73                       const Color  hintergrund = BLACK);
74  // Legt die Hintergrundfarbe und die Vordergrundfarbe fuer ein Rechteck
75  // auf dem Bildschirm fest. Das Rechteck kann aus beliebig vielen
76  // Zeichen-Zellen bestehen (maximal aus allen Zellen des Bildschirms).
77  //------------------------------------------------------------------------
78  bool Put            (const short  spalte,
79                       const short  zeile,
80                       const char  *text);
81  // Gibt den text zum Standardausgabefenster aus. Das erste Zeichen des
82  // Textes wird in die Zelle (spalte, zeile) geschrieben.
83  // ACHTUNG: Die Position des Bildschirmzeigers (cursor) wird durch dieses
84  // Unterprogramm *nicht* veraendert!
85  //------------------------------------------------------------------------
86  bool Put            (const short  spalte,
87                       const short  zeile,
88                       const char   zeichen);
89  // Gibt das zeichen zum Standardausgabefenster aus, an die Position (spalte,
90  // zeile).
91  // ACHTUNG: Die Position des Bildschirmzeigers (cursor) wird durch dieses
92  // Unterprogramm *nicht* veraendert!
93  //------------------------------------------------------------------------
94  bool GotoXY         (const short  spalte,
95                       const short  zeile);
96  // Bringt den Bildschirmzeiger im Standardausgabefenster an die Position
97  // (spalte, zeile). Nachfolgende Ausgaben (z. B. cout << "Hallo!";) erfolgen
98  // ab dieser neuen Position.
99  //------------------------------------------------------------------------
100 bool GetImmediateAny(char  *zeichen,
101                      short *psVirtualKeyCode,
```

```
102                      int   *piControlKeyState);
103 // Liest ein Zeichen von der Standardeingabe (Tastatur). Der eingebende
104 // Benutzer muss seine Eingabe *nicht* mit der Return-Taste abschliessen.
105 // Die Eingabe erfolgt *ohne Echo* (d. h. das eingegebene Zeichen erscheint
106 // nicht automatisch auf dem Bildschirm). Mit diesem Unterprogramm kann
107 // *jeder* Tastendruck ("any key") gelesen werden (auch ein Druck auf eine
108 // Taste wie Pfeil-nach-oben, Bild-nach-unten, Umschalt-Taste, linke
109 // Alt-Taste, Rechte Alt-Taste, linke Strg-Taste, rechte Strg-Taste oder
110 // auf eine Funktionstaste F1, F2 etc. etc.). Die folgenden Funktionen
111 // sollen das Analysieren des Parameters *piControlKeyState erleichtern.
112 // ------------------------------------------------------------------------
113 // Funktionen zur Ananlyse des Parameters *piControlKeyState der
114 // Funktion GetImmediateAny:
115 bool Capslock_On        (int iControlKeyState);
116 bool Enhanced_Key       (int iControlKeyState);
117 bool Left_Alt_Pressed   (int iControlKeyState);
118 bool Left_Ctrl_Pressed  (int iControlKeyState);
119 bool Numlock_On         (int iControlKeyState);
120 bool Right_Alt_Pressed  (int iControlKeyState);
121 bool Right_Ctrl_Pressed(int iControlKeyState);
122 bool Scrolllock_On      (int iControlKeyState);
123 bool Shift_Pressed      (int iControlKeyState);
124 //-------------------------------------------------------------------------
125 bool GetImmediate(char  *zeichen);
126 // Liest ein Zeichen von der Standardeingabe (Tastatur). Der eingebende
127 // Benutzer muss seine Eingabe *nicht* mit der Return-Taste abschliessen.
128 // Die Eingabe erfolgt *ohne Echo* (d. h. das eingegebene Zeichen erscheint
129 // nicht automatisch auf dem Bildschirm). Mit diesem Unterprogramm koennen
130 // nur "normale Zeichen" (Buchstaben, Ziffern, Sonderzeichen etc.) einge-
131 // lesen werden. Steuerzeichen (z. B. Pfeil-nach-oben, Bild-nach-unten etc.)
132 // werden vom Betriebssystem "verschluckt".
133 //-------------------------------------------------------------------------
134 bool FenstergroesseMaximal();
135 // Vergroessert das Standardausgabefenster auf die maximale Groesse (die
136 // von der Groesse des Bildschirms und dem verwendeten Font abhaengt).
137 //-------------------------------------------------------------------------
138 bool Fenstergroesse_80_25();
139 // Veraendert die Groesse des Standardausgabefensters so, dass es 25 Zeilen
140 // mit je 80 Spalten enthaelt.
141 //-------------------------------------------------------------------------
142 Color ColorSucc(const Color c);
143 // Zyklische Nachfolgerfunktion fuer Color (nach BRIGHT_WHITE kommt BLACK)
144 Color ColorPred(const Color c);
145 // Zyklische Vorgaengerfunktion fuer Color (vor BLACK liegt BRIGHT_WHITE)
146 const char *ColorName(const Color c);
147 // Liefert den Namen der Farbe c als String.
148 //-------------------------------------------------------------------------
149 #endif // Konsole_h
```

Die Implementierungsdatei `Konsole.cpp` enthält die *Definitionen* der in der Kopfdatei `Konsole.h` *deklarierten* Unterprogramme und wird hier nicht wiedergegeben. Mit der Quelldatei `KonsoleTst.cpp` (hier nicht wiedergegeben) und der *make*-Datei `KonsoleTst.mak` (hier nicht wiedergegeben) kann man ein Programm erzeugen, welches einem zu jedem Tastendruck auf der Tastatur den *virtual key code* etc. anzeigt.

17 Ein endlicher Automat

17.1 Problem

Es sollen Zahlen wie z. B. `+123.45` oder `-67.8` oder `901` oder `+32` etc. eingelesen werden. Das Vorzeichen (+ oder –) ist optional. Ein Dezimalpunkt ist ebenfalls optional, aber wenn er vorkommt, muss er zwischen zwei Ziffern stehen (z. B. sind `.12` oder `12.` oder `+.12` nicht erlaubt, dagegen sind `0.12` und `12.0` und `+0.12` erlaubt).

Solche Zahlen sollen mit Hilfe des Befehls `GetImmediate` aus dem Modul `Konsole` Zeichen für Zeichen eingelesen werden. Dabei sollen Zeichen, die grundsätzlich nicht erlaubt sind (z. B. ein Buchstabe wie `A` oder ein Zeichen wie `%`) ignoriert werden und keinerlei Reaktion des Programms auslösen. Zeichen, die im Prinzip erlaubt sind, aber an der falschen Stelle eingegeben werden (z. B. ein Vorzeichen nach der ersten Ziffer oder zweiter Dezimalpunkt) sollen eine Fehlermeldung auslösen.

Die vom Benutzer schon eingegebene Zeichenkette soll auf dem Bildschirm angezeigt werden, und zwar in ***grüner*** Farbe, wenn es sich um eine ***vollständige Zahl*** handelt und in ***roter*** Farbe, wenn es sich um eine noch ***unvollständige Zahl*** handelt. Z. B. sollen die Zeichenkette `+` und `+123.` in ***rot*** erscheinen, Zeichenketten wie z. B. `+1` oder `+123.4` dagegen in ***grün***.

Dieses Problem soll mit Hilfe eines ***endlichen Automaten*** gelöst werden.

17.2 Endliche Automaten

Ein ***endlicher Automat*** besteht aus drei (endlichen und nichtleeren) Mengen:
einer Menge von ***Zuständen*** {`z0`, `z1`, `z2`, ... }
einer Menge von ***Eingaben*** {`e0`, `e1`, `e2`, ... }
einer Menge von ***Aktionen*** {`a0`, `a1`, `a2`, ... }

und einer Menge von ***Regeln*** der folgenden Form:

Regel 17: Wenn der Automat sich im ***Zustand*** `z5` befindet und die nächste ***Eingabe*** gleich `e7` ist, dann wird die ***Aktion*** `a3` ausgeführt und der Automat geht in den ***Zustand*** `z6` über.

Eine solche Regel kann man kürzer auch als ein Viertupel darstellen, etwa so:

Regel 17: `(z5, e7, a3, z6)`

Ein endlicher Automat befindet sich in jedem Moment in einem bestimmten ***Zustand***. Bei jedem "Takt" liest der Automat eine ***Eingabe***, führt dann (abhängig von seinem Zustand und der Eingabe) eine ***Aktion*** aus und geht (ebenfalls abhängig von seinem Zustand und der Eingabe) in einen ***Folgezustand*** über. Dann kommt der nächste Takt, u.s.w.

17.3 Konkrete Eingabezeichen und abstrakte Eingaben

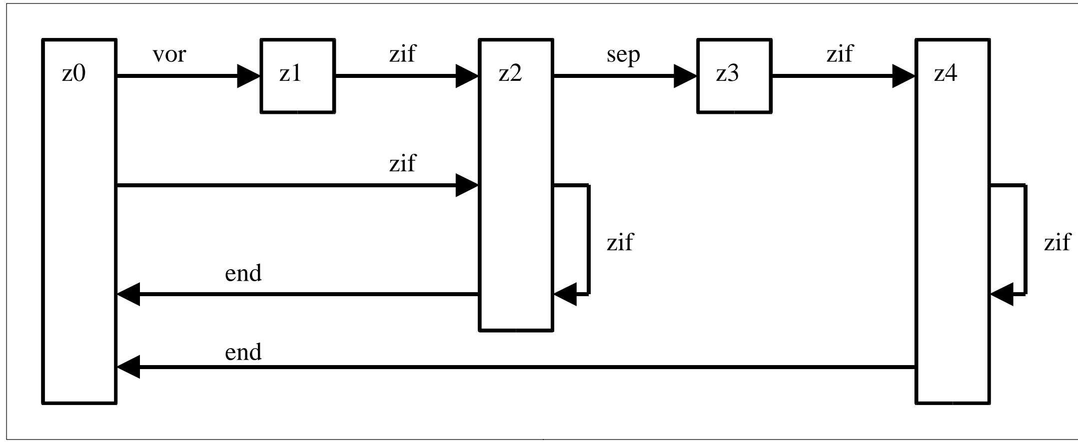
Häufig sollen verschiedene Eingabezeichen in einem Automaten dieselbe Aktion auslösen. Z. B. soll ein Pluszeichen und ein Minuszeichen genau gleich behandelt werden. Ebenso sollen alle Ziffern gleich (aber anders als die Vorzeichen) behandelt werden. Es ist deshalb häufig günstig, die ***konkreten Eingabezeichen*** eines Automaten zu Gruppen zusammenzufassen und jede Gruppe durch eine ***abstrakte Eingabe*** zu repräsentieren. Für das oben skizzierte Problem kann das z. B. so aussehen:

Konkrete Eingabezeichen	***Abstrakte Eingaben***
+ -	**vor** (wie Vorzeichen)
0 1 2 3 4 5 6 7 8 9	**zif** (wie Ziffer)
.	**sep** (wie Separator)
Blank	**end** (wie Endezeichen)

Der Folge von ***konkreten*** Eingabezeichen `+123.45` entspricht die Folge von ***abstrakten*** Eingaben `vor zif zif zif sep zif zif`. Der konkreten Eingabe `-989.99` entspricht dieselbe Folge von abstrakten Eingaben.

17.4 Der endliche Automat, dargestellt durch ein Diagramm (noch ohne Aktionen)

Der folgende endliche Automat löst das oben skizzierte Problem (Zahlen zeichenweise einlesen und prüfen). Er besitzt ***5 Zustände*** (`z0` bis `z4`) und ***4*** (abstrakte) ***Eingaben*** (`vor`, `zif`, `sep` und `end`):



Wenn dieser Automat z. B. die konkrete Zeichenfolge `+123.45` einliest, durchläuft er folgende Zustände: `z0`, `z1`, `z2`, `z2`, `z2`, `z3`, `z4`, `z4`.

In jedem ***Zustand*** werden laut diesem Diagramm nur bestimmte Eingaben ***erwartet***. Z. B. werden im Zustand `z0` die Eingaben `vor` und `zif` erwartet, nicht aber die Eingaben `sep` und `end`.

Standard-Fehlerbehandlung: Wenn in irgendeinem Zustand ein ***unerwartetes*** Zeichen eingelesen wird, dann soll eine ***Fehlermeldung*** ausgegeben, das falsche Zeichen weggeworfen und das nächste Zeichen eingelesen werden. Der ***Zustand*** des Automaten soll dabei ***nicht*** verändert werden. Standard-

Fehlerbehandlungen, die nach diesem einfachen Schema erfolgen, werden im Diagramm ***nicht*** ausdrücklich dargestellt (damit das Diagramm nicht unnötig voll und unübersichtlich wird).

Im Diagramm fehlen aber auch noch Informationen, die ergänzt werden sollten: Jede Kante im Diagramm soll nicht nur mit einer (abstrakten) ***Eingabe***, sondern zusätzlich auch mit einer ***Aktion*** beschriftet werden. Diese Aktion wird ausgeführt, wenn der Automat die entsprechende Kante "entlangläuft".

17.5 Die Aktionen des endlichen Automaten

Folgende Aktionen sind für den obigen Automaten sinnvoll:

Aktion `a1`: Akzeptiere das letzte Eingabezeichen und stelle alle bisher akzeptierten Zeichen in der ***Farbe1*** (z. B. in ***rot***) dar (weil die akzeptierten Zeichen noch keine vollständige Zahl darstellen).

Aktion `a2`: Akzeptiere das letzte Eingabezeichen und stelle alle bisher akzeptierten Zeichen in der ***Farbe2*** (z. B. in ***grün***) dar (weil die akzeptierten Zeichen eine vollständige Zahl darstellen).

Aktion `a3`: Wirf alle bisher akzeptierten Eingabezeichen weg und fange von vorn an.

Hinzu kommt noch eine Aktion für die ***Standard-Fehlerbehandlungen***, die ***nicht*** in das Diagramm des Automaten eingezeichnet werden:

Aktion `a0`: Gib eine Fehlermeldung aus, weil ein unerwartetes Zeichen eingelesen wurde.

Aufgabe: Schreiben Sie im obigen Diagramm an jede Kante eine der Aktionen `a1` bis `a3`. Welche Aktionen werden von Ihrem Automaten ausgeführt, wenn er die konkrete Eingabe `+123.45` einliest?

17.6 Der endliche Automat, dargestellt durch eine Tabelle

Endliche Automaten werden nicht nur als ***Diagramme*** dargestellt (meist für menschliche Leser), sondern auch durch ***Tabellen*** oder ***Reihungen***, die sich leichter in den Code eines Programms einbauen lassen. Hier eine Tabelle für den oben als Diagramm dargestellten Automaten:

	vor	***zif***	***sep***	***end***
z0		a2, z2		
z1				
z2				
z3				
z4				

Der Eintrag `z2, a2` (in ***Zeile*** `z0`, ***Spalte*** `zif` der Tabelle) bedeutet: Wenn der Automat sich im Zustand `z0` befindet und eine Eingabe `zif` einliest, dann führt er die Aktion `a2` aus und geht in den Zustand `z2` über.

Aufgabe: Übertragen Sie alle weiteren Informationen aus der ***Diagramm-Darstellung*** des Automaten in die ***Tabellen-Darstellung***.

Aufgabe: Wie sehen die ***Standardfehlerbehandlungen*** (die im Diagramm nicht eingezeichnet wurden) in der ***Tabellen-Darstellung*** aus?

Im C++-Programm im nächsten Abschnitt wird diese eine ***Tabelle*** durch ***zwei*** (zweidimensionale) ***Reihungen*** realisiert. Die eine Reihung enthält die ***Aktionen*** und die andere die (Folge-) ***Zustände***.

17.7 Der endliche Automat in C++ programmiert

```
1   // Datei Automat01.cpp
2   /* ------------------------------------------------------------------------
3   Ein endlicher Automat, der Zahlen einliest und prueft. Nur die 14 Zeichen
4   + - 0 1 2 3 4 5 6 7 8 9 . und Blank werden "ernst genommen", alle anderen
5   Zeichen werden ignoriert. Wenn die Zeichen + - . oder Blank "im falschen
6   Moment" eingegeben werden, erfolgt eine Fehlermeldung. Unvollstaendige Zah-
7   len wie z. B. + oder +123. werden rot, vollstaendige Zahlen wie z. B. +1 oder
8   +123.4 werden gruen dargestellt. Ein Blank veranlasst den Automaten dazu,
9   wieder von vorn anzufangen und eine neue Zahl einzulesen.
10  ---------------------------------------------------------------------- */
11  #include <iostream>
12  #include <string>
13  #include "Konsole.h" // fuer GetImmediate
14  using namespace std;
15  // ------------------------------------------------------------------------
16  // Die Zustaende und die (abstrakten) Eingaben des endlichen Automaten:
17  enum zustaende {z0, z1, z2, z3, z4};
18  enum eingaben  {vor, zif, sep, end};
19
20  int const ANZ_EINGABEN  = end + 1;
21  int const ANZ_ZUSTAENDE = z4  + 1;
22
23  // Alle Aktionen sind Unterprogramme des folgenden Typs aktion:
24  typedef void (* aktion)();
25  // ------------------------------------------------------------------------
26  // Globale Variablen
27  char         einChar;                // aktuelles konkretes Eingabezeichen
28  eingaben     aktEingabe;             // aktuelle  abstrakte Eingabe
29  zustaende    aktZustand;             // aktueller Zustand
30  aktion       aktAktion;              // aktuelle  Aktion
31  string       ausgabe;                // aktueller akzeptierter String
32  Color  const FARBE1   = BRIGHT_RED;  // fuer unvollstaendige Zahlen
33  Color  const FARBE2   = BRIGHT_GREEN;// fuer   vollstaendige Zahlen
34  Color        aktFarbe = FARBE1;      // aktuelle Farbe
35
36  short  const MAX_SPALTE = AnzSpalten()-1;
37  short  const AUS_ZEILE  = 3;         // Zeile fuer die normale Ausgabe
38  short  const ERR_ZEILE  = 5;         // Zeile fuer Fehlermeldungen
39  short  const TST_ZEILE  = 7;         // Zeile fuer Testausgabe
40  string const BLANKS(AnzSpalten(), ' ');
41  // ------------------------------------------------------------------------
42  // Drei Hilfsunterprogramme:
43  void liesZeichen() {
44     // Liest das naechste Eingabezeichen von der Standardeingabe nach einChar
45     // und bringt die entsprechende abstrakte Eingabe nach aktEingabe.
46     // Falls q oder Q eingelesen wurde, wird das Programm beendet. Grund-
47     // saetzlich nicht erlaubte Zeichen werden ignoriert ("ueberlesen").
48
49     while (true) { // Diese Schleife wird mit return oder mit exit verlassen:
50        GetImmediate(&einChar);       // ohne Echo und ohne Return einlesen
51        switch (einChar) {
52           case 'q': case 'Q':
53              cout << "\nAutomat01 beendet sich!\n"; exit(0);
54           case '+': case '-':
55              aktEingabe=vor; return; // Ein Vorzeichen wurde gelesen
56           case '0': case '1': case '2': case '3': case '4':
57           case '5': case '6': case '7': case '8': case '9':
58              aktEingabe=zif; return; // Eine Ziffer wurde gelesen
```

```
59          case '.':
60             aktEingabe=sep; return; // Ein Separator wurde gelesen
61          case ' ':
62             aktEingabe=end; return; // Ein Blank wurde gelesen
63       } // switch
64    } // while
65 } // liesZeichen
66 // ------------------------------------------------
67 void gibAus() {
68    // Gibt die ausgabe in der aktuellen Farbe in die Ausgabezeile aus:
69    SetColors(0, MAX_SPALTE, AUS_ZEILE, AUS_ZEILE, aktFarbe, BLACK);
70    Put(0, AUS_ZEILE, ausgabe.c_str());
71 } // gibAus
72 // ------------------------------------------------
73 void initAlles() {
74    // Initialisiert den endlichen Automaten:
75    ClearScreen();
76    cout << "Bitte geben Sie eine Zahl ein (oder 'q' zum Beenden oder\n"
77            "ein Blank fuer 'alles von vorn')" << endl;
78    GotoXY(0, AUS_ZEILE);  // Nur zur "Beruhigung des Benutzers"
79    aktZustand = z0;       // z0 ist der Anfangszustand des Automaten
80 } // initAlles
81 // --------------------------------------------------------------------------
82 // Die einzelnen Aktionen des endlichen Automaten:
83 void a0() {
84    // Gibt eine Fehlermeldung in die Fehlerzeile aus:
85    static const string text = "Falsches Eingabezeichen: ";
86    Put(          0, ERR_ZEILE, text.c_str());
87    Put(text.size(), ERR_ZEILE, einChar);
88 }; // a0
89 // ------------------------------------------------
90 void a1() {
91    // Akzeptiert ein Zeichen und waehlt FARBE1 aus:
92    Put(0, ERR_ZEILE, BLANKS.c_str());
93    ausgabe  += einChar;
94    aktFarbe = FARBE1;
95    gibAus();
96 } // a1
97 // ------------------------------------------------
98 void a2() {
99    // Akzeptiert ein Zeichen und waehlt FARBE2 aus:
100   Put(0, ERR_ZEILE, BLANKS.c_str());
101   ausgabe += einChar;
102   aktFarbe = FARBE2;
103   gibAus();
104 } // a2
105 // ------------------------------------------------
106 void a3() {
107   // Loescht die Ausgabezeile auf dem Bildschirm und die Variable ausgabe:
108   Put(0, ERR_ZEILE, BLANKS.c_str());
109   Put(0, AUS_ZEILE, BLANKS.c_str());
110   ausgabe = "";
111 } // a3
112 // --------------------------------------------------------------------------
113 // Der endliche Automat, dargestellt durch 2 zweidimensionale Reihungen:
114 zustaende zustandsTab[ANZ_ZUSTAENDE][ANZ_EINGABEN] = {
115   {z1, z2, z0, z0},
116   {z1, z2, z1, z1},
117   {z2, z2, z3, z0},
118   {z3, z4, z3, z3},
119   {z4, z4, z4, z0},
120 }; // zustandsTab
```

```
121 aktion aktionsTab[ANZ_ZUSTAENDE][ANZ_EINGABEN] = {
122    {a1, a2, a0, a0},
123    {a0, a2, a0, a0},
124    {a0, a2, a1, a3},
125    {a0, a2, a0, a0},
126    {a0, a2, a0, a3},
127 }; // aktionsTab
128 // ------------------------------------------------------------------------
129 // Zwei Unterprogramme die den endlichen Automaten "nachvollziehbar machen"
130 // und das Testen dieses Programms erleichtern sollen:
131 void test1() {
132    // Den alten Zustand und die Eingabe ausgeben:
133    Put   (0, TST_ZEILE, BLANKS.c_str());
134    GotoXY(0, TST_ZEILE);
135    char aktZ = static_cast<char>(aktZustand) + '0';
136    char aktE = static_cast<char>(aktEingabe) + '0';
137    cout << "Alter Zust.: z" << aktZ    << ", " <<
138            "EinChar: "      << einChar << ", " <<
139            "EinAbstrakt: "  << aktE    << ", ";
140 } // test1
141 // ---------------------------------------------
142 void test2() {
143    // Die aktuelle Aktion und den neuen Zustand ausgeben:
144    char aktZ = static_cast<char>(aktZustand) + '0';
145    char aktA                     = '0';
146    if (aktAktion == a1) aktA = '1';
147    if (aktAktion == a2) aktA = '2';
148    if (aktAktion == a3) aktA = '3';
149    SetColors();
150    cout << "Aktion: a"      << aktA  << ", " <<
151            "Neuer Zust.: z" << aktZ  << endl;
152    GotoXY(ausgabe.size(), AUS_ZEILE); // Nur zur "Beruhigung des Benutzers"
153 } // test2
154 // ------------------------------------------------------------------------
155 int main() {
156    // Der "Treiber" fuer den endlichen Automaten:
157    initAlles();
158    while (true) {
159       liesZeichen(); // In liesZeichen wird das Programm evtl. beendet!
160       test1();       // Nur zum Testen, kann auskommentiert werden!
161
162       aktAktion  = aktionsTab [aktZustand][aktEingabe];
163       aktAktion();   // Die aktuelle Aktion ausfuehren.
164       aktZustand = zustandsTab[aktZustand][aktEingabe];
165
166       test2();       // Nur zum Testen, kann auskommentiert werden!
167    } // while
168 } // main
169 // ------------------------------------------------------------------------
```

Die beiden Unterprogramme `test1` und `test2` dienen nur dazu, die Arbeitsweise des endlichen Automaten auf dem Bildschirm sichtbar und nachvollziehbar zu machen. In einer endgültigen Version des Programms sollten sie entfernt (oder ihre Aufrufe auskommentiert) werden.

Verschiedene endliche Automaten unterscheiden sich nur durch ihre *Tabellen*. Der *Treiber* (siehe Zeile 158 bis 167) ist immer der gleiche und kann im Prinzip für *beliebige endliche Automaten* verwendet werden. *Tabellengesteuerte Programme* sind im allgemeinen *leichter änderbar* als andere Programme.

18 Höchste Zeit für eine Zeitmessung

Zeitmessungen innerhalb eines Programms werden vom C++-Standard nur *schwach* unterstützt. Häufig muss man *nicht-standardisierte*, *plattformabhängige* Befehle dazu verwenden.

```
1   // Datei ZeitMessung01.cpp
2   /* ------------------------------------------------------------------------
3   Zeitmessungen in Sekunden und Millisekunden. Demonstriert werden drei
4   Vorgehensweisen (zwei davon sind Windows-spezifisch und funktionieren nicht
5   unter anderen Betriebssystemen wie Linux, OS/2, Solaris etc.)
6   ------------------------------------------------------------------------ */
7   #include <iostream>
8   #include <ctime>     // fuer time()
9   #include <windows.h> // fuer GetLocalTime, SYSTEMTIME, WORD
10                       //      GetTickCount, DWORD
11  using namespace std;
12  // ------------------------------------------------------------------------
13  // WORD und DWORD sind typedef-Namen die zum Window-API gehoeren:
14  //  WORD: 16 Bit vorzeichenlose Ganzzahl (unsigned int)
15  // DWORD: 32 Bit vorzeichenlose Ganzzahl (unsigned int)
16  // ------------------------------------------------------------------------
17  WORD GetLocalMillis() {
18     // Liefert die Anzahl der Millisekunden, die seit Anfang der laufenden
19     // Woche (d. h. seit letzten Montag um 0 Uhr frueh) vergangen sind:
20     SYSTEMTIME st;
21     DWORD       millis;
22     DWORD const msProSek = 1000;
23     DWORD const msProMin = 60 * msProSek;
24     DWORD const msProStd = 60 * msProMin;
25     DWORD const msProTag = 24 * msProStd;
26     GetLocalTime(&st);  // Systemuhr ablesen
27     millis  = st.wMilliseconds;
28     millis += st.wSecond    * msProSek;
29     millis += st.wMinute    * msProMin;
30     millis += st.wHour      * msProStd;
31     millis += st.wDayOfWeek * msProTag;
32     return millis;
33  } // GetLocalMillis
34  // ------------------------------------------------------------------------
35  int main() {
36     // Fuehrt dreimal eine triviale Schleife aus ("zum Zeit verbrauchen"),
37     // misst die dafuer benoetigte Zeit (auf drei verschiedene Weisen) und
38     // gibt die gemessenen Zeiten zur Standardausgabe aus:
39     cout << "ZeitMessung01: Jetzt geht es los!" << endl;
40     int const WIE_OFT = 1000 * 1000 * 100;  // Zum Zeit verbrauchen
41     // Zeitmessung in Sekunden mit time(0) (unabhaengig vom Betriebssystem):
42     int start1 = time(0);                   // Startzeit festhalten
43     for (int i=0; i<WIE_OFT; i++) i=2*i-i;  // Zeit verbrauchen
44     int dauer1 = time(0) - start1;          // Zeitraum seit Start berech.
45     cout << "Dauer1 (in Sekunden):      " << dauer1 << endl;
46
47     // Zeitmessung in Millisekunden mit GetLocalTime() (Windows-spezifisch):
48     DWORD start2 = GetLocalMillis();         // Startzeit festhalten
49     for (int i=0; i<WIE_OFT; i++) i=2*i-i;   // Zeit verbrauchen
50     DWORD dauer2 = GetLocalMillis() - start2;// Zeitraum seit Start berech.
51     cout << "Dauer2 (in Millisekunden): " << dauer2 << endl;
52
53     // Zeitmessung in Millisekunden mit GetTickCount() (Windows-spezifisch):
54     DWORD start3 = GetTickCount();           // Startzeit festhalten
55     for (int i=0; i<WIE_OFT; i++) i=2*i-i;   // Zeit verbrauchen
56     DWORD dauer3 = GetTickCount() - start3;  // Zeitraum seit Start berech.
57     cout << "Dauer3 (in Millisekunden): " << dauer3 << endl;
58     cout << "ZeitMessung01: Das war's erstmal!"     << endl;
59  } // main
```

```
60  /* ------------------------------------------------------------------------
61  Ausgabe des Programms ZeitMessung01 (Gnu-Cygnus-Compiler 2.95.2):
62
63  ZeitMessung01: Jetzt geht es los!
64  Dauer1 (in Sekunden):      4
65  Dauer2 (in Millisekunden): 3520
66  Dauer3 (in Millisekunden): 3486
67  ZeitMessung01: Das war's erstmal!
68  ---------------------------------------------------------------------------
69  Ausgabe des Programms ZeitMessung01 (Borland C++-Compiler 5.5):
70
71  ZeitMessung01: Jetzt geht es los!
72  Dauer1 (in Sekunden):      2
73  Dauer2 (in Millisekunden): 1260
74  Dauer3 (in Millisekunden): 1241
75  ZeitMessung01: Das war's erstmal!
76  ------------------------------------------------------------------------ */
```

19 Kommandozeilen-Befehle zum Compilieren und Binden

Programme, die nur aus ***einer*** Quelldatei bestehen, kann man im ***TextPad*** mit dem Befehl ***Compilieren und binden*** im Menü ***Extras*** erzeugen. ***Mittelgroße*** Programme kann man mit den im folgenden beschriebenen Befehlen erzeugen. Die Befehle kann man im ***TextPad*** unter ***Extras***, ***Ausführen*** oder (ohne TextPad) in ein ***Dos-Eingabeaufforderungsfenster*** eingeben. Um ***große*** Programme effizient managen zu können, sollte man sich mit dem ***make***-Programm vertraut machen.

19.1 Gnu-Cygnus-Compiler g++

Aus ***einer*** Quelldatei `dat1.cpp` eine ausführbare Datei namens `a.exe` ("Standardname") erzeugen:

```
g++ dat1.cpp
```

Aus einer Quelldatei `dat1.cpp` eine ausführbare Datei namens `otto.exe` erzeugen:

```
g++ -o otto.exe dat1.cpp
```

Aus ***mehreren*** Quelldateien eine ausführbare Datei namens `otto.exe` erzeugen:

```
g++ -o otto.exe dat1.cpp dat2.cpp dat3.cpp
```

Die ***Reihenfolge***, in der die Quelldateien angegeben werden, spielt ***keine*** Rolle.

Mehrere Quelldateien ***getrennt*** voneinander ***compilieren*** und dann die ***Objektdateien*** zu einer ausführbaren Datei namens `otto.exe` binden lassen:

```
g++ -c dat1.cpp
g++ -c dat2.cpp
g++ -c dat2.cpp
g++ -o otto.exe dat1.o dat2.o dat2.o
```

Wenn man jetzt z. B. nur an der Quelldatei `dat2.cpp` etwas ändert, muss man auch nur die folgenden Schritte wiederholen:

```
g++ -c dat2.cpp
g++ -o otto.exe dat1.o dat2.o dat2.o
```

Man kann ***Quelldateien*** und ***Objektdateien*** auch ***gemischt*** angeben, z. B. so:

```
g++ -o otto.exe dat1.o dat2.cpp dat3.o
```

Der Ausführer compiliert die Quelldatei `dat2.cpp` und bindet dann die drei Objektdatein `dat1.o`, `dat2.o` und `dat3.o` wie im vorigen Beispiel.

19.2 Borland C++Compiler bcc32

Aus ***einer*** Quelldatei `dat1.cpp` eine ausführbare Datei namens `dat1.exe` erzeugen:

```
bcc32 dat1.cpp
```

Aus einer Quelldatei `dat1.cpp` eine ausführbare Datei namens `otto.exe` erzeugen:

```
bcc32 -eotto.exe dat1.cpp
```

Zwischen dem `-e` und dem Namen `otto.exe` darf ***kein*** Blank stehen! Die Dateinamenserweiterung `.exe` muss man nicht angeben (für den `bcc32` ist sie selbstverständlich).

Aus ***mehreren*** Quelldateien eine ausführbare Datei namens `otto.exe` erzeugen:

```
bcc32 -eotto dat1.cpp dat2.cpp dat3.cpp
```

Mehrere Quelldateien ***getrennt*** voneinander ***compilieren*** und dann die ***Objektdateien*** zu einer ausführbaren Datei namens `otto.exe` binden lassen:

```
bcc32 -c dat1.cpp
bcc32 -c dat2.cpp
bcc32 -c dat3.cpp
bcc32 -eotto.exe dat1.obj dat2.obj dat3.obj
```

Man beachte, dass die ***Objektdateien*** hier die Erweiterung `.obj` (und nicht nur `.o` wie beim Gnu-Cygnus-Compiler) haben.

Man kann ***Quelldateien*** und ***Objektdateien*** auch ***gemischt*** angeben, z. B. so:

```
bcc32 -eotto.exe dat1.cpp dat2.obj dat3.cpp
```

Der Ausführer ***compiliert*** die Quelldateien `dat1.cpp` und `dat3.cpp` und ***bindet*** dann die drei Objektdateien `dat1.obj`, `dat2.obj` und `dat3.obj` wie im vorigen Beispiel.

19.3 Microsoft C++-Compiler cl

Aus ***einer*** Quelldatei `dat1.cpp` eine ausführbare Datei namens `dat1.exe` erzeugen:

```
cl /EHsc dat1.cpp
```

Aus einer Quelldatei `dat1.cpp` eine ausführbare Datei namens `otto.exe` erzeugen:

```
cl /EHsc -Feotto.exe dat1.cpp
```

Zwischen dem `-Fe` und dem Namen `otto.exe` darf ***kein*** Blank stehen! Die Dateinamenserweiterung `.exe` muss man nicht angeben (für den `cl` ist sie selbstverständlich).

Aus ***mehreren*** Quelldateien eine ausführbare Datei namens `otto.exe` erzeugen:

```
cl /EHsc -Feotto dat1.cpp dat2.cpp dat3.cpp
```

Mehrere Quelldateien ***getrennt*** voneinander ***compilieren*** und dann die ***Objektdateien*** zu einer ausführbaren Datei namens `otto.exe` binden lassen:

```
cl /EHsc -c dat1.cpp
cl /EHsc -c dat2.cpp
cl /EHsc -c dat3.cpp
cl /EHsc -Feotto.exe dat1.obj dat2.obj dat3.obj
```

Man beachte, dass die ***Objektdateien*** hier die Erweiterung `.obj` (und nicht nur `.o` wie beim Gnu-Cygnus-Compiler) haben.

Man kann ***Quelldateien*** und ***Objektdateien*** auch ***gemischt*** angeben, z. B. so:

```
bcc32 -eotto.exe dat1.cpp dat2.obj dat3.cpp
```

Der Ausführer ***compiliert*** die Quelldateien `dat1.cpp` und `dat3.cpp` und bindet dann die drei Objektdateien `dat1.obj`, `dat2.obj` und `dat3.obj` wie im vorigen Beispiel.

19.4 Ein paar Skriptdateien, zum Compilieren und Binden einzelner Quelldateien

Mit den folgenden Skriptdateien (Stapelverarbeitungsdateien, `.bat`-Dateien) kann man unter Windows eine einzelne C++-Quelldatei compilieren, binden und das erzeugte Programm starten.

```
1   @echo off
```

```
2   rem Datei bgc.bat
3   rem -----------------------------------------------------
4   rem Zum Compilieren, Binden und Ausfuehren einer C++-Datei
5   rem mit dem Gnu-C++-Compiler g++. Kopfdateien werden auch
6   rem im Boost-Verzeichnis D:/bibs/boost gesucht.
7   rem -----------------------------------------------------
8   rem Aufruf, um die Datei Hallo.cpp zu bearbeiten:
9   rem > bgc     Hallo   // In einem   Dos-Fenster
10  rem > bgc.bat Hallo   // In Cygwin-Bash-Fenster
11  rem -----------------------------------------------------
12
13  g++ -o %1.exe -ID:/bibs/boost %1.cpp
14  echo ====================================================
15  %1.exe
16  echo ====================================================
```

```
17  @echo off
18  rem Datei bcc.bat
19  rem -----------------------------------------------------
20  rem Zum Compilieren, Ausfuehren und Binden einer C++-Datei
21  rem mit dem Borland-C++-Compiler bcc32. Kopfdateien werden
22  rem auch im im Boost-Verzeichnis D:/bibs/boost gesucht.
23  rem -----------------------------------------------------
24  rem Aufruf, um die Datei Hallo.cpp zu bearbeiten:
25  rem > bcc     Hallo   // In einem   Dos-Fenster
26  rem > bcc.bat Hallo   // In Cygwin-Bash-Fenster
27  rem -----------------------------------------------------
28
29  bcc32 /ID:/bibs/boost %1
30  echo ====================================================
31  %1.exe
32  echo ====================================================
```

```
33  @echo off
34  rem Datei bcl.bat
35  rem -----------------------------------------------------
36  rem Zum Compilieren, Binden und Ausfuehren einer C++-Datei
37  rem mit dem MS-C++-Compiler cl. Kopfdateien werden auch im
38  rem Boost-Verzeichnis D:/bibs/boost gesucht.
39  rem -----------------------------------------------------
40  rem Aufruf, um die Datei Hallo.cpp zu bearbeiten:
41  rem > bcl     Hallo   // In einem   Dos-Fenster
42  rem > bcl.bat Hallo   // In Cygwin-Bash-Fenster
43  rem -----------------------------------------------------
44
45  set CC=C:\Programme\vsdn\Vc7\bin\cl.exe
46  echo Als Compiler wird aufgerufen: %CC%
47
48  %CC% /EHsc /I D:/bibs/boost %1.cpp
49  echo ====================================================
50  %1.exe
51  echo ====================================================
```

Bash-Shell-Skripte namens `b`, `bb`, `bm` (zum Compilieren und Binden einer C++-Quelldatei), `c`, `cb` und `cm` (zum Compilieren einer C++-Quelldatei) und `r` (wie run, zum Ausführen einer ausführbaren Datei) findet man bei den Beispielprogrammen. Diese Skripte kann man z. B. unter Windows in einem Cygnus-Bash-Fenster oder unter Linux in einem Bash-Fenster starten. Die Skripte sind ausführlich kommentiert.

19.5 Eine make-Datei mit ein paar Erläuterungen

Das Programm `make` ist ein Interpreter, der spezielle Skriptdateien (`make`-Dateien) ausführen kann. Mit `make`-Skripten läßt man typischerweise komplizierte Compilations-, Binde-, Installations-, Deinstallation- und Aufräumarbeiten durchführen (man soll damit aber auch Kaffeemaschinen steuern können :-). Hier ein Beispiel:

```
1   # Beispiel-make-Datei namens makefile
2   # zum Erstellen einer ausfuehrbaren Datei namens otto.exe
3   # aus den Quelldateien dat1.cpp, dat2.cpp und der Kopfdatei
4   # dat1.hpp, die von beiden Quelldateien inkludiert wird,
5   # mit dem Gnu-C++-Compiler/Binder g++
6   # Die eingerueckten Zeilen muessen mit einem Tab-Zeichen beginnen!
7
8   otto.exe: dat1.o dat2.o
9     g++ -o otto.exe dat1.o dat2.o
10
11  dat1.o: dat1.cpp dat1.hpp
12    g++ -c dat1.cpp
13
14  dat2.o: dat2.cpp dat1.hpp
15    g++ -c dat2.cpp
16
17  loesch:
18    rm otto.exe
19    rm dat1.o
20    rm dat2.o
```

Diese `make`-Datei enthaelt vier **Ziele**: `otto.exe`, `dat1.o`, `dat2.o` und `loesch`. Die ersten drei dieser Ziele bestehen darin, eine Datei zu erzeugen, das vierte Ziel ist anderer Art.

In Zeile 8 steht, dass das Ziel `otto.exe` nur erreicht werden kann, wenn schon zwei Dateien namens `dat1.o` und `dat2.o` existieren (d.h. wenn die Ziele `dat1.o` und `dat2.o` schon erreicht wurden).

In Zeile 11 steht, dass das Ziel `dat1.o` nur erreicht werden kann, wenn schon zwei Dateien namens `dat1.cpp` und `dat2.hpp` existieren. In Zeile 14 entsprechend für das Ziel `dat2.o`.

In Zeile 17 steht, dass das Ziel `loesch` von keinem anderen Ziel (von keiner Datei) abhängig ist und somit jederzeit "angegangen werden" kann.

In Zeile 9 steht, wie man das Ziel `otto.exe` erreichen kann: Indem man die Objektdateien `dat1.o` und `dat2.o` mit dem Programm `g++` zu einer ausführbaren Datei `otto.exe` zusammenbindet. Statt nur ***ein*** Kommando könnte hier auch eine beliebig lange Folge von Kommandos stehen. Falls die Datei `dat1.o` noch nicht existiert (oder älter ist als die Dateien `dat1.cpp` oder `dat1.hpp`, von denen das Ziel `dat1.o` abhängig ist, siehe Zeile 11), wird vor dem Ziel `otto.exe` das Ziel `dat1.o` angegangen. Entsprechend für `dat2.o`.

Es folgen ein paar Aufrufe des `make`-Programms:

```
21  >                           // make-Datei und Ziel
22  > make                      // makefile, erstes Ziel
23  > make -f otto.abc          // otto.abc, erstes Ziel
24  > make loesch               // makefile, Ziel loesch
25  > make dat1.o               // makefile, Ziel dat1.o
26  > make -f otto.abc loesch   // otto.abc, Ziel loesch
27  > make -f otto.abc dat1.o   // otto.abc, Ziel dat1.o
```

Das `make`-Programm geht immer das erste oder das angegebene Ziel an. Falls dieses "Oberziel" von anderen "Unterzielen" abhängt, geht `make` erstmal diese Unterziele an (und falls die von Unterunterzielen abhängen, dann erstmal diese Unterunterziele etc.).

20 Sachwortverzeichnis

_unexpected_handler 17, 19f.
- (Sub.op.) 124, 143
:: (scope op.) 117, 158f.
* (Var.op.) 59
& (Adressop.) 59
#include-Bef. 25
+ (Add.op.) 124, 143
<< (Ausgabeop.) 15
<algorithm> 81
<cfloat> 37
<climits> 10, 14, 16, 37
<cmath> 129
<cstddef> 50, 52, 63, 69
<cstdlib> 81
<ctime> 169
<exception> 17, 19
<fstream> 98ff., 104
<iomanip> 37f., 81, 84, 98, 106f.
<iostream> 98, 161
<limits> 38
<sstream> 109f., 129
<string> 106
<strstream.h> 109f.
<vector> 77
<windows.h> 169
>> (Eingabeop.) 102
A
abstrakte
  Klassen 136
  Methoden 136, 141
Ada 123, 147
adjustfield 107ff.
Adress-
  konstante 61
  operator & 59, 66
  typ 59, 62f.
  variable 41, 44, 62f.
  wert 62
Adressen
  rechnen mit 69
Algol68 39
Algorithmen 81, 83, 87
Alias 73
anonymous
  namespace 150
Anweisung 7
  break- 13
  catch- 17
  continue- 13
  do-while- 10
  einfache- 10
  for- 10f., 70
  if- 10, 65
  switch- 10f.
  throw- 17
  try- 17
  while- 10f.
  zusamm.ges.- 10
append 84
array 49
Assembler 8
assign 84
at 84
Attribut 113
Aufgaben 14, 16, 24, 30, 47f., 52, 72, 75, 90, 100f., 126, 140, 165
Aufzählungstypen 35f., 160
Ausdruck 7
Ausdrucks-if 65
Ausführer 7
Ausgabe
  -Breite 31, 107f.
  -Operator 15, 120, 124f.
  -Strom 97
ausgeben
  formatiert 98f.
  Reihungen 50f.
  unformatiert 99
  Vectoren 15, 79, 93
Ausnahme 17, 149
  erwartete- 17
  geprüfte- 17
  unerwartete- 17
  ungeprüfte- 17
  werfen 17
Automat
  Aktionen eines 163
  Eingaben eines 163
  Zustände eines 163
B
basefield 107f.
Bauplan
  für Module 7, 111
  für Objekte 112
  für Variablen 7
Bauplanaspekt 111, 113, 117
bcc32, Compiler 171
Beerbungsgraph 132, 137
Behälter
  in C++ 81, 145
  in Java 145
Beispielprogramme
  Adressen01 59
  Adressen25 59
  Algorithmen01 81
  Ausnahmen01 17
  Ausnahmen02 18
  Ausnahmen03 19f.
  Automat01 166
  Birnen 123ff.
  BirnenTst 125
  CppStrings01 83
  CStrings01 56
  CStrings02 58
  DekDef01 26, 28
  EinAus01 106
  EinAus10 99
  EinAus11 100, 102
  EinAus13 102
  EinAus14 107
  EinAus16 110
  Einlesen04 104
  Figuren01 129
  FundTypen02 37
  FundTypen03 38
  Hallo01 9
  Hallo02 9
  Hallo03 9
  Hallo04 10
  Haupt 23f.
  IfSwitch01 10
  Klassen10 111
  Klassen11 118f.
  Klassen12 120
  Konsole 160, 162
  KonsoleTst 162
  KonstruierteTypen01 45ff.
  KonstruierteTypen02 47
  Mehrfach01 132

Mehrfach02 134f.
Mehrfach03 138f.
Mehrfach04 138f.
NamensRaeume01 156
NamensRaeume03 157
NamensRaeume04 158
NamensRaeume05 158
Neben 23
Obst 142, 144
ObstTst 144f.
Referenz01 59
Referenz04 59
Reihungen02 52, 54
Reihungen04 54
Reihungen10 50f.
Reihungen12 54
Schablonen01 89f.
Schablonen02 91f.
Schablonen03 93
Schablonen05 94
Schablonen07 94
Schleifen01 11f.
Schleifen02 13
StapelK_Tst 153
StapelM_Tst 150
StapelS_Tst 155
Static01.cpp 34
Static02a.cpp 147
Static02b.cpp 147
Upro01 13f.
Upro02 14
Vector01 78
Vector02 80f.
Virtuell01 127
Virtuell02 136f.
WarteTicket 114ff.
WarteTicket_Tst 114, 116
Zeitmessung 169
benutzen, Größe 7, 29
Benutzer 7
Beton 49, 77
Binder (linker) 24f., 32, 174
Birnen 123
Bojen 40, 55
Boost-Bibliothek 95
Borland 6, 38
break-Anweis. 13
Breite 31, 107f.
Breymann 5
Bruchzahlen 31, 107f.
Bruchzahltyp 94
bubblesort 15
buffer overflow 50
C
C-Strings 56
C, die Sprache 8
C++-Reihungen 77
C++-Standard 5f., 8
C++-Strings 83
catch-Anweis. 17
char *, char [] 56
CHAR_MIN 37
cin 9, 97
cin >> s 102
cin.get() 102
cl, Compiler 172
clear 81, 98
Compilationsmodell
von C++ 24
von Java 24
Compiler
Borland 171
Gnu-Cygnus 171
Microsoft 172
Compiler Collection 6
Compilezeit 87f., 94
const 71, 75
const_iterator 80
container
in C++ 145
in Java 145
continue-Anw. 13
cout 9, 97
CR 103f., 106
Cygnus 6, 171
D
Datei 21
-Umlenkung 97
ausführbare- 24, 171f.
Heimat- 147
make- 174
Quell- 22, 24
Datenquelle 97
Datensenke 97
DBL_MIN 37
declaration 7
default arguments 13
definieren
ein Unterprog. 26
eine Klasse 114
eine Konst. 26f.
eine Variable 26f.
definierte Typen 36
Definition
in Datei 26
in Klasse 114
Deitel 6
Deklaration
in Datei 26
in Klasse 114
in Namensr. 158
deklarieren
ein Unterprog. 26
eine Klasse 114
eine Konst. 26
eine Variable 26
deque 79, 83, 94, 145
Dereferenz. 59
Destruktor 120, 151ff.
do-while-Anw. 10
dynamisch 22, 126f.
E
Effizienz 8, 49, 126
einfaches Erben 126, 129
Eingaben 163
Eingabestrom 97f., 100f., 104, 106
einlesen 102
Element 113
endlich. Automat 168
enum type 35f., 160
erase 81
Erben
einfaches- 126, 129
mehrfaches- 132, 134, 137
Erzeugungsbefehl 21
exception 17
exe-Datei 24
Exponent 12f., 98, 107
expression 7
externe Darstellung 99
F
field 113
floatfield 107

FLT_MIN 37
flush 98
for_each 81
for-Anweis. 10f., 70
Formatfehler 98
formatieren
  mit printf 31
formatierte E/A 99
function template 87, 89, 94f.
fundamental 35, 37, 45
funktionale Spr. 96
G
g++, Compiler 6, 171
GanzNachString 94
Ganzzahl 13, 123, 142
Gedächtnis
  eines Unterprog. 34
Generische Einh. 87
getline 104
GetLocalTime 169
GetTickCount 169
Gnu-Cygnus 6, 171
Grabo 8, 145, 160
Graph
  zyklenfreier 140
Größe 7
  benutzen 29
  UV- 26
Gummi 49, 77
H
Hardware
  -Schwächen 123
Hauptklasse 22
header file 29
Heimatdatei 147
I
IEC 6, 8, 71
if-Anweis. 10, 65
if-Operator 65
include-Bef. 25
Indexprüfung
  mit 78, 84
  ohne 78, 84
indirection 59, 68
Inprise 6
insert 80, 85
Instanzen
  einer Schablone 77, 87
INT_MIN 37
interne Darstellung 99, 109
ISO 6, 8, 71
Iterator 79ff.
J
Java 14
  -Programm 22
  abstrak. Meth. 136, 141
  Ausnahmen 17
  Behälter 145
  Bojen 45
  collection 145
  Compilationsmodell 24
  container 145
  erben 132, 140
  generische E. 146
  interface 141
  Klassen 140
  konstr. Typen 36
  main-Meth. 22
  Methoden 13
  Pakete 9, 21, 156
  Reihungen 54
  Sammlung 145
  Schnittst. 141
  Schwäche 123
  String 83
  Typen 35, 45
  und C++ 5
  Vector 77
Josuttis 5
K
Kernighan 5
Klassen 111
  -Attribut 112ff., 117
  -Element 43, 113, 124
  -Methode 22, 114, 117, 138
  definieren 114
  deklarieren 114
Klausel, throw- 17
Knoten
  einer Liste 43
Kollegen 7
Komponenten 42, 49, 61
  -Typ 91
  einer Queue 83
  einer Reihung 15, 42, 50
  eines Strings 84
konkatenieren 56, 84, 87
konsistent 25
Konsole 160
Konstante 7, 27, 60
  CHAR_MIN 37
  DBL_MIN 37
  definieren 26
  deklarieren 26
  FLT_MIN 37
  INT_MIN 14, 37
  LONG_MIN 37
  SHRT_MIN 37
konstruierte Typen 36, 45, 47
Konstruktor 118, 120f., 153
Kopfdateien 29
  <algorithm> 81
  <cfloat> 37
  <climits> 10, 14, 16, 37
  <cmath> 129
  <cstddef> 50, 52, 63, 69
  <cstdlib> 81
  <ctime> 169
  <exception> 17, 19
  <fstream> 98ff., 104
  <iomanip> 37f., 81, 84, 98, 106f.
  <iostream> 98, 161
  <limits> 38
  <sstream> 109f., 129
  <string> 106
  <strstream.h> 109f.
  <vector> 77
  <windows.h> 169
  Konsole.h 160
  Obst.h 142
  StapelK.h 151
  StapelM.h 149
  StapelS.h 153f.
Kopier-Konstruktor 120ff.
L
L-Wert 40, 64
Lajoie 6, 74
Laufzeit 88, 94, 146
Länge
  einer Reihung 49
  eines C-String 56
  logische 56

physikalische 56
Leung 6
LF 103f., 106
Linker (binder) 24f., 32, 174
Linux 6, 160, 169, 173
Lippman 6, 74
Lischner 6
Literal 60
logische, Länge 56
LONG_MIN 37
M
Macintosh 104
main 9, 21
make 174
Manipulator 98, 107
resetiosflags 107
setfill 107
setiosflags 107
setw 107
mehrdimens.
Reihung 52, 54f.
mehrfache Vererbung 132
mehrfaches Erben 132, 134, 137
mehrstufige
Reihung 54f.
Metaprogrammierung 94
Methode
einer Klasse 113
Modul 7, 111, 147
Heimat- 147
Modulaspekt 111, 113
N
Nachpunktstellen 31f., 107
name resolution 92
Name, Alias- 73
namenlos 40, 42, 148, 150
Namensauflösung 92
Namenskonkflikt 22
Namensraum 21f.
geschachtelt.- 158
globaler 21
mit Namen 9, 21, 156
namenloser 150
namespace
anonymous 150
NaN 62f., 123, 134
NanA 56, 62ff.
Nappo-Frage 137
NULL 56, 63
Null-begrenzte 56
Null-Byte 56
O
Objektattribut 113
Objektdatei 24f., 171f., 174
Objektelement 111, 115f.
Objektmethode 114, 117, 141
Obst.h 142
ODR, die Regel 26
one definition rule 26
Operation 13
Operator 13, 15, 124
- 124, 143
* 59
& 59
+ 124, 143
<< 15
>> 102
Adress- 66
Deref.- 59
if- 65
Indirekt.- 59
Variablen- 66
OS/2 160, 169
P
Parameter
Schablonen- 87ff., 142, 146, 153
Typ- 87
Unterprog.- 91
per Referenz 76
physikalische
Länge 56
Plattform 104, 123
pointer 59
pop_back 81
Potenzieren 87, 89
Präprozessor 25
precision 75, 84, 107
preprocessor 25
printf 31
privat 27, 43, 113, 147
Gedächtnis 34
Programmierer 7
Programmierung
Meta- 94
Prozedur 7
prozedurale Spr. 95
public 43
Pufferüberlauf 50
Punktnotation 117
pure virtual 136, 141
push_back 79
Q
Quelldatei 22, 24
R
R-Wert 40, 64
rand 82
rechnen
mit Adressen 69
mit Ganzahl. 123
reference types 35f., 41
Referenz 128
Referenz-Var. 41
Referenztypen 35f., 41, 59, 73f.
in C++ 41, 59
in Java 41, 59
Reihung 49
Bojen 40, 55
eindimens. 50
mehrdimens. 52, 54f., 165
mehrstufige 54f.
von char 56
Zuweisung an eine 51
Reihungsproblem 49
rein virtuell 136, 141
reinterpret_cast 53, 101f.
remove_if 81
replace 83, 85
resetiosflags 107f.
reverse_iterator 80
Ritchie 5
Rollenspiel 7
runden 108
S
scanf 33
Schablone 87, 142
Schablone
Funktions- 87, 89, 94f.
Klassen- 77, 142
Schablonenpar. 87ff., 142, 146, 153
Schlüsselwort 3, 11, 17, 34,

43, 45, 137, 139, 147f.
scientific 107
seekg 104
set_unexpected 17, 19f.
setfill 107f.
setiosflags 107f.
setw 106
showpos 107f.
SHRT_MIN 37
Sichtbarkeit 117
sizeof 49, 56
Solaris 160, 169
sort 81
sortieren 14
sprintf 33
sscanf 33
Standard
-Ausgabe 9, 97
-Bibliothek 5, 8, 77, 79, 97
-Eingabe 9, 97
-Konstruktor 120f., 123f.
C++- 5f., 8
StapelK.h 151
StapelM.h 149
StapelS.h 153f.
statement 7
static 34, 147
static_cast 19, 168
statisch 126f.
STL 81, 145
strcat 56
strcmp 56
strcpy 56
stream 97f., 100f., 104, 106
String 56, 83
Null-begrenzter 56
StringNachGanz 94
strlen 56
Strom 97
Strom-Methoden 107
Strom, Ausgabe- 97
Stroustrup 5
Ströme
verknüpfen 98
switch-Anweis. 10f.
T
tabellengesteuert 168
template 87, 142
class- 77, 142
function- 87, 89, 94f.
Teufel 104
throw-Anweis. 17
throw-Klausel 17
tie, streams 98
time 169
Tondo 6
toString 94, 129ff.
transform 81
transparente Z. 103f., 106
try-Anweis. 17
typedef-Dekl. 26, 28, 169
Typen 35
-System 35, 87
Adress- 59, 62
Aufzählungs- 35f., 160
definierte- 36
konstruierte- 36, 45, 47
Pointer- 59
Referenz- 35f., 41, 59, 73f.
Reihungs- 49
Unterprog.- 35f., 166
Zeiger- 59
Typparameter 87, 91
U
überladen 13, 91, 134
Überlauf 13
Puffer- 50
überschreiben 134
Unendlich 123
unerwartet 19
unexpected 19f.
unexpected_handler 17, 19f.
unformat. E/A 99
Unix 24, 97, 104
Unruh, Erwin 94
Unterprog.-Param. 91
Unterprogramm 7, 13, 16
als Parameter 3, 13f., 81
definieren 26
deklarieren 26
VAP- 16
uppercase 107f.
using-Deklarat. 22, 156f.
using-Direktive 9
UV-Größe 26
V
VAP-Unterprog. 16
Variable 7, 27, 40, 60
Adress- 41, 44, 62
definieren 26
deklarieren 26
Referenz- 41
Variablenop. * 59, 66
vector 77, 81
Vereinbarung 7, 21
Vererbung
mehrfache- 132
verkettete Liste 43
verknüpfen
Ströme 98
Versprechen 23, 26
vertausche 67, 74
virtuelle Meth. 126
rein- 136, 141
Vorbesetzung 13
W
WarteTicket.h 114ff.
Weiss 5
werfen, Ausnahm. 17
Wert
Adress- 62
NaN- 62f., 123, 134
NanA- 62ff.
Wertebehälter 7
while-Anweis. 10f.
wiederverwend. 147
Z
Zeiger 59
Zeitmessung 169
Zufallszahlen 82
Zustände 163
Zuweisung 51
Zuweisungs-Op. 64, 120, 123
Zweierkomplement 123
zyklenfrei 140